



C/C++ Language Reference



C/C++ Language Reference

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 439.

Second Edition (October 2001)

This edition applies to Version 1, Release 2, Modification 0, of z/OS C/C++ (program 5694-A01), and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces *ILE C for AS/400 Language Reference*, SC09-2711-01, *VisualAge C++ for OS/400 C++ Language Reference*, SC09-2121-00, *OS/390 C/C++ Language Reference*, SC09-2360-05, and *z/OS V1R1 C/C++ Language Reference*, SC09-4764-00. Make sure that you use the correct edition for the level of the program listed above. Also, ensure that you apply all necessary PTFs for the program.

This book contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line (|) to the left of the change.

You may notice changes in the style and structure of some content in this book; for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our books.

Order publications through your IBM representative or the IBM branch office serving your location. Publications are not stocked at the address below. You can also browse the books on the World Wide Web by clicking on "The Library" link on the z/OS home page. The web address for this page is
<http://www.ibm.com/servers/eserver/zseries/zos/bkserv>

IBM welcomes your comments. You can send your comments in any one of the following methods:

- Electronically to the network ID listed below. Be sure to include your entire network address if you want a reply.

Internet: torrcf@ca.ibm.com

- By mail, to the following address:

IBM Canada Ltd. Laboratory
Information Development
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

If you send comments, include the title and order number of this book, and the page number or topic related to your comment. When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1998, 2001. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Reference	xiii
z/OS C/C++ and Related Publications	xiii
Highlighting Conventions	xiii
How to Read the Syntax Diagrams	xiii
A Note About Examples	xv
 z/OS C/C++: A Platform-Specific Summary	xvii
z/OS Language Environment Downward Compatibility	xvii
The z/OS C/C++ Compilers	xviii
The C Language	xviii
The C++ Language	xviii
Common Features of the z/OS C and C++ Compilers	xviii
z/OS C Compiler Specific Features	xx
z/OS C++ Compiler Specific Features	xx
Class Libraries	xx
IBM Open Class Library Source	xxi
Utilities	xxi
The Debug Tool	xxii
IBM C/C++ Productivity Tools for z/OS	xxii
z/OS Language Environment	xxiii
The Program Management Binder	xxiv
z/OS UNIX System Services (z/OS UNIX)	xxiv
z/OS C/C++ Applications with z/OS UNIX C/C++ Functions	xxvi
Input and Output	xxvi
I/O Interfaces	xxvi
File Types	xxvii
Additional I/O Features	xxviii
The System Programming C Facility	xxviii
Interaction with Other IBM Products	xxviii
Additional Features of z/OS C/C++	xxx
Related Publications	xxx
Softcopy Books	xxxii
z/OS C/C++ on the World Wide Web	xxxii
 Chapter 1. Scope and Linkage	1
Scope	1
Local Scope	1
Function Scope	2
Function Prototype Scope	2
Global Scope	2
Class Scope	3
Example of Scope in C	3
Name Hiding	4
Program Linkage	5
Internal Linkage	5
External Linkage	6
No Linkage	7
Linkage Specifications — Linking to Non-C++ Programs	7
Name Spaces of Identifiers	8
 Chapter 2. Lexical Elements	11
Tokens	11
Source Program Character Set	11

Escape Sequences	13
The Unicode Standard	14
Trigraph Sequences	15
Digraph Characters	16
Comments	16
Identifiers	18
Case Sensitivity and Special Characters in Identifiers.	19
Significant Characters in Identifiers	19
Keywords	20
Alternative Tokens.	21
z/OS C/C++ External Name Mapping.	22
z/OS Long Name Support	22
Literals	23
Integer Literals	24
Floating-Point Literals	25
Fixed-Point Decimal Constants (z/OS C Only)	27
Character Literals	28
String Literals	29
Boolean Literals	31
Chapter 3. Declarations	33
Declarations Overview	33
Objects.	34
Storage Class Specifiers	34
auto Storage Class Specifier	35
extern Storage Class Specifier	37
extern Storage Class Specifier (z/OS)	38
Controlling External Static (z/OS)	39
Examples of extern Storage Class (z/OS)	40
mutable Storage Class Specifier	40
register Storage Class Specifier.	41
static Storage Class Specifier	42
typedef.	43
Type Specifiers.	44
Simple Type Specifiers	44
Structures.	51
Unions	59
Enumerations	65
volatile and const Qualifiers	69
Incomplete Types	71
Chapter 4. Declarators	73
_Packed Qualifier (z/OS C Only)	74
__cdecl Keyword (z/OS C++ Only)	75
Semantics of __cdecl	76
Examples of __cdecl Use	76
_Export Keyword (z/OS C++ only).	77
Initializers	79
C/C++ Data Mapping (z/OS)	80
Pointers	81
Declaring Pointers.	81
Assigning Pointers	82
Initializing Pointers	82
Restrictions on Pointers	83
Restrictions on z/OS C Pointers	83
Using Pointers	84

Pointer Arithmetic	84
Example Program Using Pointers	85
Arrays	86
Declaring Arrays	86
Initializing Arrays	88
Example Programs Using Arrays	90
Function Specifiers	92
References	92
Initializing References	93
Chapter 5. Expressions and Operators	95
Operator Precedence and Associativity	95
Examples of Expressions and Precedence.	98
Lvalues and Rvalues.	99
Type-Based Aliasing	100
Integer Constant Expressions	101
Primary Expressions	101
Parenthesized Expressions ()	101
C++ Scope Resolution Operator ::	102
Postfix Expressions.	103
Function Calls ()	104
Array Subscript [] (Array Element Specification)	106
Dot Operator	107
Arrow Operator ->	107
static_cast Operator	108
reinterpret_cast Operator.	109
const_cast Operator	110
dynamic_cast Operator	111
Unary Expressions	113
Increment ++	114
Decrement --	114
Unary Plus +	115
Unary Minus -	115
Logical Negation !	115
Bitwise Negation ~	116
Address &	116
Indirection *.	117
sizeof (Size of an Object)	117
digitsof and precisionof (z/OS C Only)	118
C++ new Operator	119
C++ delete Operator	122
Allocation and Deallocation Functions	123
Binary Expressions	124
Multiplication *.	125
Division /	126
Remainder %	126
Addition +	126
Subtraction -	127
Bitwise Left and Right Shift << >>	127
Relational < > <= >=	128
Equality == !=	129
Bitwise AND &	130
Bitwise Exclusive OR ^	130
Bitwise Inclusive OR 	131
Logical AND &&	132
Logical OR 	132

C++ Pointer to Member Operators <code>.*</code> <code>->*</code>	133
Assignment Expressions	134
Cast Expressions	135
C++ throw Expressions	137
Conditional Expressions	137
Type of Conditional C Expressions	137
Type of Conditional C++ Expressions	138
Examples of Conditional Expressions	138
The typeid Operator	139
Comma Expression ,	140
Chapter 6. Implicit Type Conversions	143
Integral and Floating-Point Promotions.	143
Standard Type Conversions.	144
Lvalue-to-Rvalue Conversions	145
Boolean Conversions	145
Integral Conversions	145
Signed-Integer Conversions (z/OS)	146
Floating-Point Conversions	146
Pointer Conversions	146
Reference Conversions	147
Pointer-to-Member Conversions	148
Qualification Conversions	148
Function Argument Conversions	148
Other Conversions	149
Arithmetic Conversions	149
The explicit Keyword	150
Chapter 7. Functions.	153
C++ Enhancements to C Functions	153
Function Declarations	154
C++ Function Declarations	155
Examples of Function Declarations	157
Function Definitions.	158
Ellipsis and void	161
Examples of Function Definitions	161
The main() Function	162
Arguments to main	163
Example of Arguments to main	163
Calling Functions and Passing Arguments	164
Command-Line Arguments (z/OS)	165
Passing Arguments by Value	167
Passing Arguments by Reference	167
Default Arguments in C++ Functions	169
Restrictions on Default Arguments	170
Evaluating Default Arguments	171
Function Return Values	171
Using References as Return Types	172
Pointers to Functions	173
Inline Functions	174
Chapter 8. Statements	177
Labels	177
Expression Statements	178
Resolving Ambiguous Statements in C++.	178
Block Statement	179

if Statement	180
switch Statement.	182
while Statement	186
do Statement	187
for Statement	188
break Statement	190
continue Statement.	190
return Statement.	192
Value of a return Expression and Function Value	192
goto Statement	193
Null Statement	194
 Chapter 9. Preprocessor Directives	 195
Preprocessor Overview	195
Preprocessor Directive Format.	196
Macro Definition and Expansion (#define)	196
Object-Like Macros	196
Function-Like Macros	197
Scope of Macro Names (#undef)	199
# Operator	200
Macro Concatenation with the ## Operator	201
Preprocessor Error Directive (#error)	202
File Inclusion (#include)	202
ISO Standard Predefined Macro Names	204
z/OS C/C++ Predefined Macro Names.	205
Examples of z/OS Predefined Macros	212
Conditional Compilation Directives	213
#if, #elif	215
#ifdef	215
#ifndef	216
#else	216
#endif	217
Examples of Conditional Compilation Directives	217
Line Control (#line)	218
Null Directive (#)	219
Pragma Directives (#pragma)	219
z/OS Pragma Directives	220
Restrictions on z/OS #pragma Directives	222
IPA Considerations	224
chars	224
checkout.	225
comment	225
convlit.	226
csect	227
define (z/OS C++ Only)	229
disjoint	229
enum	230
environment (z/OS C Only)	231
export.	232
filetag	232
implementation (z/OS C++ Only)	233
info (z/OS C++ Only)	233
inline (z/OS C Only)	234
isolated_call	236
langlvl.	237
leaves.	238

linkage (z/OS C only)	239
longname	241
map	242
margins	243
namemangling (z/OS C++ only)	245
noinline	245
object_model (z/OS C++ Only)	246
options (z/OS C Only)	247
option_override	247
pack	249
page (z/OS C Only)	250
pagesize (z/OS C Only)	251
priority (z/OS C++ Only)	251
reachable	252
report (z/OS C++ only)	252
runopts	253
sequence	254
skip (z/OS C Only)	255
strings	256
subtitle (z/OS C Only)	256
target (z/OS C Only)	256
title (z/OS C Only)	257
variable	257
wsizeof	258

Chapter 10. Namespaces	261
Defining Namespaces	261
Declaring Namespaces	261
Creating a Namespace Alias	261
Creating an Alias for a Nested Namespace	262
Extending Namespaces	262
Namespaces and Overloading	263
Unnamed Namespaces	264
Namespace Member Definitions	265
Namespaces and Friends	265
Using Directive	266
The using Declaration and Namespaces	267
Explicit Access	267

Chapter 11. Overloading	269
Overloading Functions	269
Restrictions on Overloaded Functions	270
Overloading Operators	271
Overloading Unary Operators	273
Overloading Binary Operators	274
Overloading Assignments	274
Overloading Function Calls	276
Overloading Subscripting	277
Overloading Class Member Access	278
Overloading Increment and Decrement	278
Overload Resolution	280
Implicit Conversion Sequences	280
Resolving Addresses of Overloaded Functions	282

Chapter 12. Classes	283
Declaring Class Types	283

Using Class Objects	284
Classes and Structures	286
Scope of Class Names	287
Incomplete Class Declarations	288
Nested Classes	288
Local Classes	290
Local Type Names	291
Chapter 13. Class Members and Friends	293
Class Member Lists	293
Data Members	294
Member Functions	295
const and volatile Member Functions	296
Virtual Member Functions	296
Special Member Functions	297
Member Scope	297
Pointers to Members	298
The this Pointer	300
Static Members	303
Using the Class Access Operators with Static Members	303
Static Data Members	304
Static Member Functions	306
Member Access	308
Friends	310
Friend Scope	312
Friend Access	314
Chapter 14. Inheritance	315
Derivation	317
Inherited Member Access	320
Protected Members	320
Access Control of Base Class Members	321
The using Declaration and Class Members	322
Overloading Member Functions from Base and Derived Classes	324
Changing the Access of a Class Member	325
Multiple Inheritance	327
Virtual Base Classes	328
Multiple Access	329
Ambiguous Base Classes	329
Virtual Functions	333
Ambiguous Virtual Function Calls	337
Virtual Function Access	338
Abstract Classes	339
Chapter 15. Special Member Functions	341
Constructors and Destructors Overview	341
Constructors	342
Default Constructors	343
Explicit Initialization with Constructors	344
Initializing Base Classes and Members	346
Construction Order of Derived Class Objects	349
Destructors	350
Free Store	353
Temporary Objects	357
User-Defined Conversions	358
Conversion by Constructor	360

Conversion Functions	361
Copy Constructors	362
Copy Assignment Operators	363
Chapter 16. Templates	367
Template Parameters	368
Type Template Parameters	368
Non-Type Template Parameters	369
Template Template Parameters	369
Default Arguments for Template Parameters.	370
Template Arguments	370
Template Type Arguments	371
Template Non-Type Arguments	371
Template Template Arguments.	373
Class Templates	374
Class Template Declarations and Definitions	376
Static Data Members and Templates	377
Member Functions of Class Templates.	377
Friends and Templates	378
Function Templates	379
Template Argument Deduction	380
Overloading Function Templates	385
Partial Ordering of Function Templates.	386
Template Instantiation	387
Implicit Instantiation.	387
Explicit Instantiation.	389
Template Specialization	390
Explicit Specialization	390
Partial Specialization	395
Name Binding and Dependent Names	397
The Keyword typename	398
The Keyword template as Qualifier	399
Chapter 17. Exception Handling	401
The try Keyword	401
Nested Try Blocks	403
catch Blocks	403
Function try block Handlers	404
Arguments of catch Blocks	407
Matching between Exceptions Thrown and Caught	407
Order of Catching	408
The throw Expression	409
Rethrowing an Exception.	410
Stack Unwinding	411
Exception Specifications	412
Special Exception Handling Functions	415
unexpected()	415
terminate()	416
set_unexpected() and set_terminate()	418
Example of Using the Exception Handling Functions	418
Appendix A. C and C++ Compatibility on the z/OS Platform	421
Constructs Found in Both C++ and C	421
Character Array Initialization	421
Class and typedef Names	421
Class and Scope Declarations.	421

const Object Initialization	422
Definitions	422
Definitions within Return or Argument Types	422
Enumerator Type	422
Enumeration Type	422
Function Declarations	422
Functions with an Empty Argument List	422
Global Constant Linkage	422
Jump Statements	423
Keywords	423
main() Recursion.	423
Names of Nested Classes	423
Pointers to void	423
Prototype Declarations	423
Return without Declared Value.	423
__STDC__ Macro	423
typedefs in Class Declarations.	423
Interactions with Other Products	424
Appendix B. Common Usage C Language Level for the z/OS Platform	425
Appendix C. Conforming to POSIX 1003.1	427
Appendix D. Supporting ISO Standards	429
Identifiers	429
Characters	430
String Conversion	430
Integers	431
Floating-Point	431
Arrays and Pointers.	432
Registers	432
Structures, Unions, Enumerations, Bit Fields	432
Declarators	433
Statements	433
Preprocessing Directives	433
Library Functions	433
CCNRABG	434
Error Handling.	434
Signals	435
Translation Limits	435
Streams, Records, and Files	436
Memory Management	437
Environment	437
Localization.	438
Time	438
Notices	439
Programming Interface Information	440
Trademarks and Service Marks	441
Industry Standards	442
Index	443

About This Reference

► **C** The C language documented in this reference is consistent with the one described in *Programming languages – C* (ISO/IEC 9899:1990).

► **C++** The C++ language documented in this reference is consistent with the one described in *Programming languages – C++* (ISO/IEC 14882:1998).

► **z/OS** This book also contains descriptions of the IBM C language and C++ language definitions that comply with the POSIX and XPG4 standards and that are implemented by the z/OS Language Environment.

z/OS C/C++ and Related Publications

This section lists the major publications of the z/OS C/C++ documentation. For a task-based presentation of information across the entire documentation set, please consult the Related Publications section in any of the following publications.

- *z/OS C/C++ Programming Guide*
- *z/OS C/C++ User's Guide*
- *z/OS C/C++ Run-Time Library Reference*
- *z/OS C/C++ Compiler and Run-Time Migration Guide*
- *z/OS C/C++ Reference Summary*

Highlighting Conventions

Bold	Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.
<i>Italics</i>	Identify parameters whose actual names or values are to be supplied by the programmer. <i>Italics</i> are also used for the first mention of new terms that are defined in the glossary.
Example	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code, messages from the system, or information that you should actually type.

How to Read the Syntax Diagrams

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

The ►— symbol indicates the beginning of a command, directive, or statement.

The —► symbol indicates that the command, directive, or statement syntax is continued on the next line.

The ►— symbol indicates that a command, directive, or statement is continued from the previous line.

The —► symbol indicates the end of a command, directive, or statement.

Diagrams of syntactical units other than complete commands, directives, or statements start with the ►— symbol and end with the —► symbol.

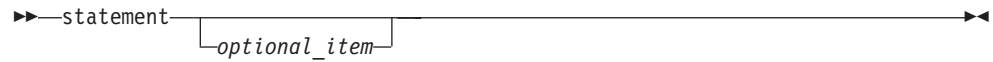
Note: In the following diagrams, statement represents a C or C++ command, directive, or statement.

Reading the Syntax Diagrams

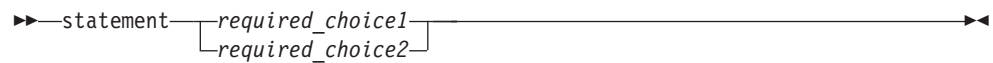
- Required items appear on the horizontal line (the main path).



- Optional items appear below the main path.



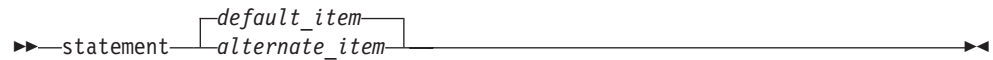
- If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.



If choosing one of the items is optional, the entire stack appears below the main path.



The item that is the default appears above the main path.



- An arrow returning to the left above the main line indicates an item that can be repeated.



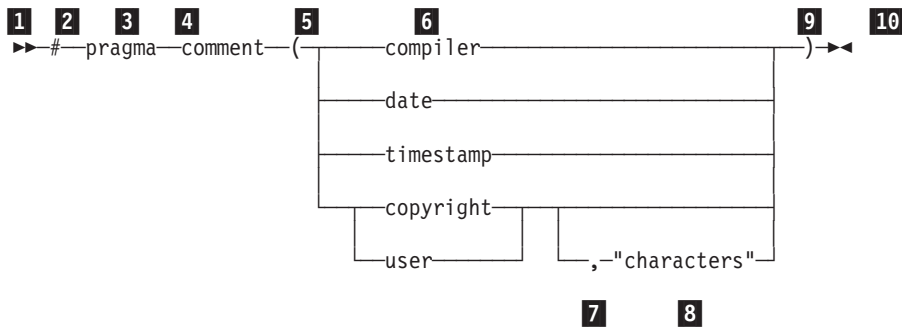
A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, `extern`).

Variables appear in italicized lowercase letters (for example, *identifier*). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

The following syntax diagram example shows the syntax for the **#pragma comment** directive. See “Pragma Directives (#pragma)” on page 219 for information on the **#pragma** directive.



- 1 This is the start of the syntax diagram.
- 2 The symbol # must appear first.
- 3 The keyword pragma must appear following the # symbol.
- 4 The name of the pragma comment must appear following the keyword pragma.
- 5 An opening parenthesis must be present.
- 6 The comment type must be entered only as one of the types indicated: compiler, date, timestamp, copyright, or user.
- 7 A comma must appear between the comment type copyright or user, and an optional character string.
- 8 A character string must follow the comma. The character string must be enclosed in double quotation marks.
- 9 A closing parenthesis is required.
- 10 This is the end of the syntax diagram.

The following examples of the **#pragma comment** directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

A Note About Examples

The examples that illustrate the use of the C/C++ compiler use a simple style. They are instructional examples, and do not attempt to minimize run time, conserve storage, or check for errors. The examples do not demonstrate all of the uses of C/C++ language constructs. Some examples are only code fragments, and will not compile without additional code.

Many of the larger examples in this book are available in machine-readable form. A label on an example indicates that the example is distributed in softcopy. The label is the name of a member of the data sets CBC.SCCNSAM or the directory /usr/lpp/ioclib/sample. The labels have the form CCNxxyy or CLBxxyy, where x refers to a publication:

- R and X refer to *C/C++ Language Reference*, SC09-4815
- G refers to *z/OS C/C++ Programming Guide*, SC09-4765
- U refers to *z/OS C/C++ User's Guide*, SC09-4767
- A refers to *IBM Open Class Library User's Guide*, SC09-2363

Examples labeled as CCNxxyy appear in *C/C++ Language Reference*, *z/OS C/C++ Programming Guide*, and *z/OS C/C++ User's Guide*.

Reading the Syntax Diagrams

z/OS C/C++: A Platform-Specific Summary

The C/C++ feature of the IBM® z/OS licensed program provides support for C and C++ application development and deployment on the z/OS platform. z/OS V1R2 C/C++ is the latest in a family of IBM C and C++ compilers for IBM mainframes, which began in 1988 with C/370™ V1R1. z/OS V1R2 C/C++ is a follow-on to OS/390® V2R10 C/C++, and contains ISO C++ language support, which is compatible with VisualAge® C++ Professional for AIX, Version 5.0.

z/OS C/C++ includes:

- A C compiler (referred to as the z/OS C compiler)
- A C++ compiler (referred to as the z/OS C++ compiler)
- Support for a set of C++ class libraries that are available with the base z/OS operating system
- Application Support Class and Collection Class Library source
- A mainframe interactive Debug Tool (optional)
- Performance Analyzer host component, which supports the C/C++ Productivity Tools for z/OS product
- A set of utilities for C/C++ application development

IBM offers the C language on other platforms, such as the AIX®, OS/400®, VM/ESA®, and VSE/ESA™ operating systems. The AIX and OS/400 operating systems also offer the C++ language.

z/OS Language Environment Downward Compatibility

z/OS Language Environment® provides downward compatibility support. Assuming that you have met the required programming guidelines and restrictions, described in *z/OS Language Environment Programming Guide*, this support enables you to develop applications on higher release levels of z/OS for use on platforms that are running lower release levels of z/OS or OS/390. In C and C++, downward compatibility support is provided through the C/C++ TARGET compiler option. See *z/OS C/C++ User's Guide* for details on this compiler option.

For example, a company may use z/OS V1R2 with Language Environment on a development system where applications are coded, link-edited, and tested, while using any supported lower release of OS/390 or z/OS Language Environment on their production systems where the finished application modules are used.

Downward compatibility support is not the roll-back of new function to prior releases of the operating system. Applications developed that exploit the downward compatibility support must not use any Language Environment function that is unavailable on the lower release of OS/390 or z/OS where the application will be used.

The downward compatibility support includes toleration PTFs for lower releases of OS/390 or z/OS to assist in diagnosing applications that do not meet the programming requirements for this support. (Specific PTF numbers can be found in the PSP buckets.)

The downward compatibility support provided by z/OS Language Environment and by the toleration PTFs does not change Language Environment's upward compatibility. That is, applications coded and link-edited with one release of OS/390 or z/OS Language Environment will continue to run on later releases of OS/390 or

z/OS Language Environment without the need to recompile or re-link edit the application, independent of the downward compatibility support.

Downward compatibility is supported in earlier releases of OS/390 C/C++ (from Version 2 Release 6), but in OS/390 V2R6, the user is required to copy header files and link-edit syslib data sets from the deployment release of OS/390. Starting with OS/390 Version 2 Release 10, the current level header files and syslib can be used (the user no longer has to copy header files and syslib data sets from the deployment release).

The z/OS C/C++ Compilers

The following sections describe the C and C++ languages and the z/OS C/C++ compilers.

The C Language

The C language is a general purpose, versatile, and functional programming language that allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages. It also provides many of the benefits of a low-level language.

The C++ Language

The C++ language introduces classes, which are user-defined data types that may contain data definitions and function definitions. You can use classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can inherit properties from one or more classes. Not only do classes describe the data types and functions available, but they can also hide (encapsulate) the implementation details from user programs. An object is an instance of a class.

The C++ language also provides templates and other features that include access control to data and functions, and better type checking and exception handling. It also supports polymorphism and the overloading of operators.

Common Features of the z/OS C and C++ Compilers

The C and C++ compilers, when used with z/OS Language Environment, offer many features to help your work:

- Optimization support:
 - Algorithms to take advantage of the S/390® architecture to get better optimization for speed and use of computer resources through the OPTIMIZE and IPA compiler options.
 - The OPTIMIZE compiler option, which instructs the compiler to optimize the machine instructions it generates to produce faster- running object code to improve application performance at run time.
 - Interprocedural Analysis (IPA), to perform optimizations across compilation units, thereby optimizing application performance at run time.
- DLLs (dynamic link libraries) to share parts among applications or parts of applications, and dynamically link to exported variables and functions at run time. DLLs allow a function reference or a variable reference in one executable to use a definition located in another executable at run time. You can use both load-on-reference and load-on-demand DLLs. When your program refers to a function or variable which resides in a DLL, z/OS C/C++ generates code to load

the DLL and access the functions and variables within it. This is called *load-on-reference*. Alternatively, your program can use z/OS C library functions to load a DLL and look up the address of functions and variables within it. This is called *load-on-demand*. Your application code explicitly controls load-on-demand DLLs at the source level.

You can use DLLs to split applications into smaller modules and improve system memory usage. DLLs also offer more flexibility for building, packaging, and redistributing applications.

- Full program reentrancy.

With reentrancy, many users can simultaneously run a program. A reentrant program uses less storage if it is stored in the LPA (link pack area) or ELPA (extended link pack area) and simultaneously run by multiple users. It also reduces processor I/O when the program starts up, and improves program performance by reducing the transfer of data to auxiliary storage. z/OS C programmers can design programs that are naturally reentrant. For those programs that are not naturally reentrant, C programmers can use constructed reentrancy. To do this, compile programs with the RENT option and use the program management binder supplied with z/OS or the z/OS Language Environment Prelinker (prelinker) and program management binder. The z/OS C++ compiler always ensures that C++ programs are reentrant.

- Inline compiler option.

Additional optimization capabilities are available with the INLINE compiler option.

- Locale-based internationalization support derived from *IEEE POSIX 1003.2-1992* standard. Also derived from *X/Open CAE Specification, System Interface Definitions, Issue 4* and *Issue 4 Version 2*. This allows programmers to use locales to specify language/country characteristics for their applications.
- The ability to call and be called by other languages such as assembler, COBOL, PL/1, compiled Java™, and Fortran, to enable programmers to integrate z/OS C/C++ code with existing applications.
- Exploitation of z/OS and z/OS UNIX technology.

z/OS UNIX is an IBM implementation of the open operating system environment, as defined in the XPG4 and POSIX standards.

- Support for the following standards at the system level:

- A subset of the extended multibyte and wide character functions as defined by *Programming Language C Amendment 1*. This is *ISO/IEC 9899:1990/Amendment 1:1994(E)*
- *ISO/IEC 9945-1:1990(E)/IEEE POSIX 1003.1-1990*
- A subset of *IEEE POSIX 1003.1a, Draft 6, July 1991*
- *IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2*
- A subset of *IEEE POSIX 1003.4a, Draft 6, February 1992* (the IEEE POSIX committee has renumbered POSIX.4a to POSIX.1c)
- *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*
- A subset of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, as applicable to the z/Series environment.
- *X/Open CAE Specification, Network Services, Issue 4*

- Year 2000 support
- Support for the Euro currency.

z/OS C Compiler Specific Features

In addition to the features common to z/OS C and C++, the z/OS C compiler provides you with the following capabilities:

- The ability to write portable code that supports the following standards:
 - All elements of the ISO standard *ISO/IEC 9899:1990 (E)*
 - *ANSI/ISO 9899:1990[1992]* (formerly *ANSI X3.159-1989 C*)
 - *X/Open Specification Programming Language Issue 3, Common Usage C*
 - *FIPS-160*
- System programming capabilities, which allow you to use z/OS C in place of assembler
- Extensions of the standard definitions of the C language to provide programmers with support for the z/OS environment, such as fixed-point (packed) decimal data support

z/OS C++ Compiler Specific Features

In addition to the features common to z/OS C and C++, the z/OS C++ compiler supports the *International Standard for the C++ Programming Language (ISO/IEC 14882-1998)* specification.

Class Libraries

z/OS V1R2 C/C++ provides the following class libraries, which are all thread-safe:

- C++ Standard Library, including the Standard Template Library (STL) and other library features of ISO C++ 1998
- IBM Open Class[®] Library for z/OS V1R2
- IBM Open Class Library for OS/390 V2R10

Refer to *z/OS C/C++ Compiler and Run-Time Migration Guide* and *IBM Open Class Library User's Guide* for more details on the components of these libraries.

For new code and the most portable code you will want to use the new C++ Standard Library, which includes the following:

- The C++ Standard I/O Stream Library for performing input and output (I/O) operations
- The C++ Standard Complex Mathematics Library for manipulating complex numbers
- The Standard Template Library (STL), which is composed of C++ template-based algorithms, container classes, iterators, localization objects, and the string class

The IBM Open Class (IOC) is a comprehensive library of C++ classes that you can use to develop applications. z/OS V1R2 includes a new level of IOC that is consistent with that which shipped in VisualAge C++ for AIX V5.0. This is intended to ease porting from AIX, but is not intended for use in new development. Support will be withdrawn in a future release.

The z/OS V1R2 IBM Open Class Library includes:

- The Application Support Class Library, which provides the basic abstractions that are needed during the creation of most C++ applications, including String, Date, Time, and Decimal. The Application Support Class Library corresponds to the IOC member in the data sets.
- The Collection Class Library implements a wide variety of classical data structures such as stack, tree, list, hash table, and so on. The Collection Class

Library provides developers with a consistent set of building blocks from which they can derive application objects. The library design exploits features of the C++ language such as exception handling and template support. The Collection Class Library corresponds to the COLL member in the data sets.

The z/OS V1R2 IBM Open Class enables you to choose between the C++ Standard I/O Stream and Complex Mathematics libraries, and the UNIX Systems Laboratories C++ Language System Release (USL) I/O Stream and Complex Mathematics libraries.

The OS/390 V2R10 IBM Open Class Library and USL libraries include the following:

- The USL I/O Stream Class Library (corresponds to the IOSTREAM member in the data sets)
- The USL Complex Mathematics Class Library (corresponds to the COMPLEX member in the data sets)
- The Application Support Class Library (corresponds to the APPSUPP member in the data sets)
- The Collection Class Library (corresponds to the COLLECT member in the data sets)

Note: Retroactive to OS/390 Version 1 Release 3, the IBM Open Class Library is licensed with the base operating system. This enables applications to use this library at run time without having to license the z/OS C/C++ compiler features or to use the DLL Rename Utility.

IBM Open Class Library Source

The IBM Open Class Library Source consists of the following:

- Application Support Class Library source code
- Collection Class Library source code

Utilities

The z/OS C/C++ compilers provide the following utilities:

- The CXXFILT utility to map z/OS C++ mangled names to the original source.
- The DSECT Conversion Utility to convert descriptive assembler DSECTs into z/OS C/C++ data structures.
- The localedef utility to read the locale definition file and produce a locale object that the locale-specific library functions can use.
- The makedepend utility to derive all dependencies in the source code and write these into the makefile for the make command to determine which source files to recompile, whenever a dependency has changed. This frees the user from manually monitoring such changes in the source code.

z/OS Language Environment provides the following utilities:

- The Object Library Utility (C370LIB) to update partitioned data set (PDS and PDS/E) libraries of object modules and Interprocedural Analysis (IPA) object modules.
- The DLL Rename Utility to make selected DLLs a unique component of the applications with which they are packaged. The DLL Rename Utility does not support XPLINK.

- The prelinker which combines object modules that comprise an z/OS C/C++ application, to produce a single object module. The prelinker supports only object and extended object format input files, and does not support GOFF.

The Debug Tool

z/OS C/C++ supports program development by using the Debug Tool. This optionally available tool allows you to debug applications in their native host environment, such as CICS/ESA®, IMS/ESA®, DB2®, and so on. The Debug Tool provides the following support and function:

- Step mode
- Breakpoints
- Monitor
- Frequency analysis
- Dynamic patching

You can record the debug session in a log file, and replay the session. You can also use the Debug Tool to help capture test cases for future program validation or to further isolate a problem within an application.

You can specify either data sets or hierarchical file system (HFS) files as source files.

Note: You can also use the dbx shell command to debug programs, as described in *z/OS UNIX System Services Command Reference*, SA22-7802.

For further information, see “IBM C/C++ Productivity Tools for z/OS”.

IBM C/C++ Productivity Tools for z/OS

With the *IBM C/C++ Productivity Tools for OS/390* product, you can expand your z/OS application development environment out to the workstation, while remaining close to your familiar host environment. IBM C/C++ Productivity Tools for OS/390 includes the following workstation-based tools to increase your productivity and code quality:

- A Performance Analyzer to help you analyze, understand, and tune your C and C++ applications for improved performance
- A Distributed Debugger that allows you to debug C or C++ programs from the convenience of the workstation
- A workstation-based editor to improve the productivity of your C and C++ source entry
- Advanced online help, with full text search and hypertext topics as well as printable, viewable, and searchable Portable Document Format (PDF) documents

In addition, IBM C/C++ Productivity Tools for OS/390 includes the following host components:

- Debug Tool
- Host Performance Analyzer

Use the Performance Analyzer on your workstation to graphically display and analyze a profile of the execution of your host z/OS C or C++ application. Use this information to time and tune your code so that you can increase the performance of your application.

Use the Distributed Debugger to debug your z/OS C or C++ application remotely from your workstation. Set a break point with the simple click of the mouse. Use the windowing capabilities of your workstation to view multiple segments of your source and your storage, while monitoring a variable at the same time.

Use the workstation-based editor to quickly develop C and C++ application code that runs on z/OS. Context-sensitive help information is available to you when you need it.

References to *Performance Analyzer* in this document refer to the IBM Performance Analyzer included in the IBM C/C++ Productivity Tools for OS/390 product.

z/OS Language Environment

z/OS C/C++ exploits the C/C++ run-time environment and library of run-time services available with z/OS Language Environment (formerly OS/390 Language Environment, Language Environment for MVS™ & VM, Language Environment/370 and LE/370).

z/OS Language Environment consists of four language-specific run-time libraries, and Base Routines and Common Services, as shown below. z/OS Language Environment establishes a common run-time environment and common run-time services for language products, user programs, and other products.

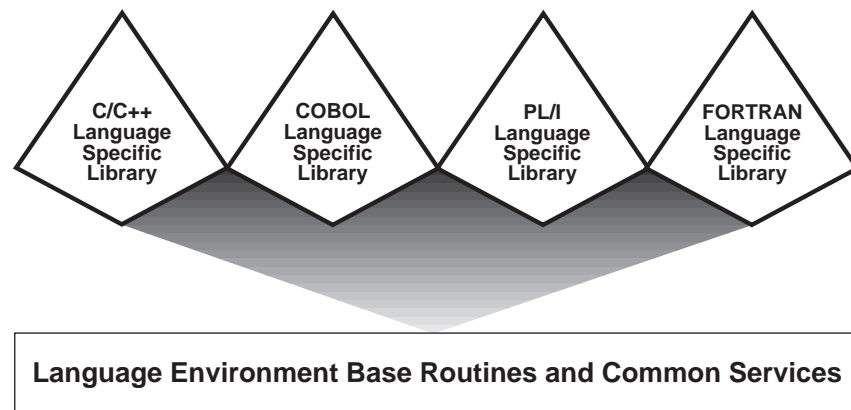


Figure 1. Libraries in z/OS Language Environment

The common execution environment is composed of data items and services that are included in library routines available to an application that runs in the environment. The z/OS Language Environment provides a variety of services:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, interlanguage communication (ILC), and condition handling.
- Extended services that are often needed by applications. z/OS C/C++ contains these functions within a library of callable routines, and includes interfaces to operating system functions and a variety of other commonly used functions.
- Run-time options that help in the execution, performance, and diagnosis of your application.
- Access to operating system services. z/OS UNIX services are available to an application programmer or program through the z/OS C/C++ language bindings.

- Access to language-specific library routines, such as the z/OS C/C++ library functions.

The Program Management Binder

The binder provided with z/OS combines the object modules, load modules, and program objects comprising an application. It produces a single z/OS output program object or load module that you can load for execution. The binder supports all C and C++ code, provided that you store the output program in a PDSE (Partitioned Data Set Extended) member or an HFS file.

If you cannot use a PDSE member or HFS file, and your program contains C++ code, or C code that is compiled with any of the RENT, LONGNAME, DLL, or IPA compiler options, you must use the prelinker. C and C++ code compiled with the GOFF or XPLINK compiler options cannot be processed by the prelinker.

Using the binder without using the prelinker has the following advantages:

- Faster rebinds when recompiling and rebinding a few of your source files
- Rebinding at the single compile unit level of granularity (except when you use the IPA compile-time option)
- Input of object modules, load modules, and program objects
- Improved long name support:
 - Long names do not get converted into prelinker generated names
 - Long names appear in the binder maps, which enables full cross-referencing
 - Variables do not disappear after prelink
 - Fewer steps in the process of producing your executable program

Using the binder without using the prelinker has the following disadvantage:

- Long name maximum symbol length:
 - Long names currently processed by the binder are limited to 1024 characters. The prelinker supports up to (32 K - 1) characters. IBM intends to bring the binder limit in line with the prelinker in a future release.

The prelinker provided with z/OS Language Environment combines the object modules that make up a z/OS C/C++ application and produces a single object module. You can link-edit the object module into a load module, which is stored in a PDS, or bind it into a load module or a program object that is stored in a PDS, PDSE, or HFS file.

Note: For further information on the binder, refer to the DFSMS home page at <http://www.ibm.com/storage/software/sms/smshome.htm>.

z/OS UNIX System Services (z/OS UNIX)

z/OS UNIX provides capabilities under z/OS to make it easier to implement or port applications in an open, distributed environment. z/OS UNIX Services are available to z/OS C/C++ application programs through the C/C++ language bindings available with z/OS Language Environment.

Together, the z/OS UNIX System Services, z/OS Language Environment, and z/OS C/C++ compilers provide an application programming interface that supports industry standards.

z/OS UNIX provides support for both existing z/OS applications and new z/OS UNIX applications through the following:

- C programming language support as defined by ISO C
- C++ programming language support as defined by ISO C++
- C language bindings as defined in the IEEE 1003.1 and 1003.2 standards; subsets of the draft 1003.1a and 1003.4a standards; X/Open CAE Specification: System Interfaces and Headers, Issue 4, Version 2, which provides standard interfaces for better source code portability with other conforming systems; and X/Open CAE Specification, Network Services, Issue 4, which defines the X/Open UNIX[®] descriptions of sockets and X/Open Transport Interface (XTI)
- z/OS UNIX Extensions that provide z/OS–specific support beyond the defined standards
- The z/OS UNIX Shell and Utilities feature, which provides:
 - A shell, based on the Korn Shell and compatible with the Bourne Shell
 - A shell, tcsh, which is based on the C shell csh
 - Tools and utilities that support the *X/Open Single UNIX Specification*, also known as *X/Open Portability Guide (XPG) Version 4, Issue 2*, and provide z/OS support. The following is a partial list of utilities that are included:

ar	Creates and maintains library archives
BPXBATCH	Allows you to submit batch jobs that run shell commands, scripts, or z/OS C/C++ executable files in HFS files from a shell session
c89	Compiles, assembles, and binds z/OS UNIX C/C++ and assembler applications
dbx	Provides an environment to debug and run programs
gencat	Merges the message text source files message file (usually *.msg) into a formatted message catalog file (usually *.cat)
iconv	Converts characters from one code set to another
lex	Automatically writes large parts of a lexical analyzer based on a description that is supplied by the programmer
localedef	Creates a compiled locale object
make	Helps you manage projects that contain a set of interdependent files, such as a program with many z/OS source and object files, by keeping all such files up to date with one another
yacc	Allows you to write compilers and other programs that parse input according to strict grammar rules
 - Support for other utilities such as:

c++	Compiles, assembles, and binds z/OS UNIX C++ applications
mkcatdefs	Preprocesses a message source file for input to the gencat utility
runcat	Invokes mkcatdefs and pipes the message catalog source data (the output from mkcatdefs) to gencat
dspcat	Displays all or part of a message catalog
dspmsg	Displays a selected message from a message catalog
- The z/OS UNIX Debugger feature, which provides the dbx interactive symbolic debugger for z/OS UNIX applications

- Access to a hierarchical file system (HFS), with support for the POSIX.1 and XPG4 standards
- z/OS C/C++ I/O routines, which support using HFS files, standard z/OS data sets, or a mixture of both
- Application threads (with support for a subset of POSIX.4a)
- Support for z/OS C/C++ DLLs

z/OS UNIX offers program portability across multivendor operating systems, with support for POSIX.1, POSIX.1a (draft 6), POSIX.2, POSIX.4a (draft 6), and XPG4.2.

To application developers who have worked with other UNIX environments, the z/OS UNIX Shell and Utilities are a familiar environment for C/C++ application development. If you are familiar with existing MVS development environments, you may find that the z/OS UNIX environment can enhance your productivity. Refer to *z/OS UNIX System Services User's Guide* for more information on the Shell and Utilities.

z/OS C/C++ Applications with z/OS UNIX C/C++ Functions

All z/OS UNIX C functions are available at all times. In some situations, you must specify the `POSIX(ON)` run-time option. This is required for the POSIX.4a threading functions and the `system()` and signal handling functions, where the behavior is different between POSIX/XPG4 and ISO. Refer to *z/OS C/C++ Run-Time Library Reference* for more information about requirements for each function.

You can invoke a z/OS C/C++ program that uses z/OS UNIX C functions by using the following methods:

- Directly from a shell.
- From another program, or from a shell, using one of the `exec` family of functions, or the `BPXBATCH` utility from TSO or MVS batch.
- Using the `POSIX system()` call.
- Directly through TSO or MVS batch without the use of the intermediate `BPXBATCH` utility. In some cases, you may require the `POSIX(ON)` run-time option.

Input and Output

The C/C++ run-time library that supports the z/OS C/C++ compiler supports different input and output (I/O) interfaces, file types, and access methods. The C++ I/O Stream Class Library provides additional support.

I/O Interfaces

The C/C++ run-time library supports the following I/O interfaces:

C Stream I/O

This is the default and the ISO-defined I/O method. This method processes all input and output by character.

Record I/O

The library can also process your input and output by record. A record is a set of data that is treated as a unit. It can also process VSAM data sets by record. Record I/O is a z/OS C/C++ extension to the ISO standard.

TCP/IP Sockets I/O

z/OS UNIX provides support for an enhanced version of an

industry-accepted protocol for client/server communication that is known as *sockets*. A set of C language functions provides support for z/OS UNIX sockets. z/OS UNIX sockets correspond closely to those used by UNIX applications that conform to the Berkeley Software Distribution (BSD) 4.3 standard (also known as OE sockets). The slightly different interface of the X/Open CAE Specification, Networking Services, Issue 4, is supplied as an additional alternative. This interface is known as X/Open Sockets.

The z/OS UNIX socket application program interface (API) provides support for both UNIX domain sockets and Internet domain sockets. UNIX domain sockets, or *local sockets*, allow interprocess communication within z/OS independent of TCP/IP. Local sockets behave like traditional UNIX sockets and allow processes to communicate with one another on a single system. With Internet sockets, application programs can communicate with others in the network using TCP/IP.

In addition, the C++ I/O Stream Library supports formatted I/O in C++. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. This helps improve the maintainability of programs that use input and output.

File Types

In addition to conventional files such as sequential files and partitioned data sets, the C/C++ run-time library supports the following file types:

Virtual Storage Access Method (VSAM) Data Sets

z/OS C/C++ has native support for three types of VSAM data organization:

- Key-sequenced data sets (KSDS). Use KSDS to access a record through a key within the record. A key is one or more consecutive characters that are taken from a data record that identifies the record.
- Entry-sequenced data sets (ESDS). Use ESDS to access data in the order it was created (or in the reverse order).
- Relative-record data sets (RRDS). Use RRDS for data in which each item has a particular number (for example, a telephone system with a record associated with each number).

For more information on how to perform I/O operations on these VSAM file types, see the *z/OS C/C++ Programming Guide*.

Hierarchical File System Files

z/OS C/C++ recognizes Hierarchical File System (HFS) file names. The name specified on the `fopen()` or `freopen()` call must conform to certain rules. You can create regular HFS files, special character HFS files, or FIFO HFS files. You can also create links or directories.

Memory Files

Memory files are temporary files that reside in memory. For improved performance, you can direct input and output to memory files rather than to devices. Since memory files reside in main storage and exist only while the program is executing, you use them primarily as work files. You can access memory files across load modules through calls to non-POSIX `system()` and `C fetch()`; they exist for the life of the root program. Standard streams can be redirected to memory files on a non-POSIX `system()` call using command line redirection.

Hiperspace™ Expanded Storage

Large memory files can be placed in Hiperspace expanded storage to free up some of your home address space for other uses. Hiperspace expanded

storage or high performance space is a range of up to 2 gigabytes of contiguous virtual storage space. A program can use this storage as a buffer (1 gigabyte equals 2^{30} bytes).

Additional I/O Features

z/OS C/C++ provides additional I/O support through the following features:

- Large file support, which allows I/O to and from data files larger than 2 gigabytes
- User error handling for serious I/O failures (SIGIOERR)
- Improved sequential data access performance through enablement of the DFSMS/MVS® support for 31-bit sequential data buffers and sequential data striping on extended format data sets
- Full support of PDSEs on z/OS, including support for multiple members opened for write
- Overlapped I/O support under z/OS (NCP, BUFNO)
- Multibyte character I/O functions
- Fixed-point (packed) decimal data type support in formatted I/O functions
- Support for multiple volume data sets that span more than one volume of DASD or tape
- Support for Generation Data Group I/O

The System Programming C Facility

The System Programming C (SPC) facility allows you to build applications that require no dynamic loading of z/OS Language Environment libraries. It also allows you to tailor your application to better utilize the low-level services that are available on your operating system. SPC offers a number of advantages:

- You can develop applications that you can execute in a customized environment rather than with z/OS Language Environment services. Note that if you do not use z/OS Language Environment services, only some built-in functions and a limited set of C/C++ run-time library functions will be available to you.
- You can substitute the z/OS C language in place of assembler language when writing system exit routines, by using the interfaces that are provided by SPC.
- SPC lets you develop applications featuring a user-controlled environment, in which an z/OS C environment is created once and used repeatedly for C function execution from other languages.
- You can utilize co-routines by using a two-stack model to write application service routines. In this model, the application calls on the service routine to perform services independently of the user. The application is then suspended when control is returned to the user application.

Interaction with Other IBM Products

When you use z/OS C/C++, you can write programs that utilize the power of other IBM products and subsystems:

- Cross System Product (CSP)
Cross System Product/Application Development (CSP/AD) is an application generator that provides ways to interactively define, test, and generate application programs to improve productivity in application development. Cross System Product/Application Execution (CSP/AE) takes the generated program and executes it in a production environment.

Note: You cannot compile CSP applications with the z/OS C++ compiler. However, your z/OS C++ program can use interlanguage calls (ILC) to call z/OS C programs that access CSP.

- Customer Information Control System (CICS®)

You can use the CICS/ESA Command-Level Interface to write C/C++ application programs. The CICS Command-Level Interface provides data, job, and task management facilities that are normally provided by the operating system.

Note: Code preprocessed with CICS/ESA versions prior to V4 R1 is not supported for z/OS C++ applications. z/OS C++ code preprocessed on CICS/ESA V4 R1 cannot run under CICS/ESA V3 R3.

- DB2 Universal Database™ (UDB) for z/OS

DB2 programs manage data that is stored in relational databases. You can access the data by using a structured set of queries that are written in Structured Query Language (SQL).

The DB2 program uses SQL statements that are embedded in the program. The SQL translator (DB2 preprocessor) translates the embedded SQL into host language statements that perform the requested functions. The z/OS C/C++ compilers compile the output of the SQL translator. The DB2 program processes a request, and processing returns to the application.

- Data Window Services (DWS)

The Data Window Services (DWS) part of the Callable Services Library allows your C or C++ program to manipulate temporary data objects that are known as TEMPSPACE and VSAM linear data sets.

- Information Management System (IMS)

The Information Management System/Enterprise Systems Architecture (IMS/ESA) product provides support for hierarchical databases.

- Interactive System Productivity Facility (ISPF)

z/OS C/C++ provides access to the Interactive System Productivity Facility (ISPF) Dialog Management Services. A dialog is the interaction between a person and a computer. The dialog interface contains display, variable, message, and dialog services as well as other facilities that are used to write interactive applications.

- Graphical Data Display Manager (GDDM)

GDDM® provides a comprehensive set of functions to display and print applications most effectively:

- A windowing system that the user can tailor to display selected information
- Support for presentation and keyboard interaction
- Comprehensive graphics support
- Fonts, including support for double-byte character set (DBCS)
- Business image support
- Saving and restoring graphic pictures
- Support for many types of display terminals, printers, and plotters

- Query Management Facility (QMF)

z/OS C supports the Query Management Facility (QMF), a query and report writing facility, which allows you to write applications through a callable interface. You can create applications to perform a variety of tasks, such as data entry, query building, administration aids, and report analysis.

- z/OS Java Support

The Java language supports the Java Native Interface (JNI) for making calls to and from C/C++. These calls do not use ILC support but rather the Java defined interface JNI. Java code, which has been compiled using the High Performance Compiler for Java (HPCJ), will support the JNI interface. There is no distinction between compiled Java and interpreted Java as far as calls to C or C++.

Additional Features of z/OS C/C++

Feature	Description
long long Data Type	The z/OS C/C++ compiler supports long long as a native data type when the compiler option <code>LANGVL(LONGLONG)</code> is turned on. This option is turned on by default by the compiler option <code>LANGVL(EXTENDED)</code> .
Multibyte Character Support	z/OS C/C++ supports multibyte characters for those national languages such as Japanese whose characters cannot be represented by a single byte.
Wide Character Support	Multibyte characters can be normalized by z/OS C library functions and encoded in units of one length. These normalized characters are called wide characters. Conversions between multibyte and wide characters can be performed by string conversion functions such as <code>wcstombs()</code> , <code>mbstowcs()</code> , <code>wcsrtombs()</code> , and <code>mbsrtowcs()</code> , as well as the family of wide-character I/O functions. Wide-character data can be represented by the <code>wchar_t</code> data type.
Extended Precision Floating-Point Numbers	<p>z/OS C/C++ provides three z/Series floating-point number data types: single precision (32 bits), declared as <code>float</code>; double precision (64 bits), declared as <code>double</code>; and extended precision (128 bits), declared as <code>long double</code>.</p> <p>Extended precision floating- point numbers give greater accuracy to mathematical calculations.</p> <p>As of Release 6, z/OS C/C++ also supports IEEE 754 floating-point representation. By default, <code>float</code>, <code>double</code>, and <code>long double</code> values are represented in IBM z/Series floating point format. However, the IEEE 754 floating-point representation is used if you specify the <code>FLOAT(IEEE754)</code> compiler option. For details on this support, see the description of the <code>FLOAT</code> option in <i>z/OS C/C++ User's Guide</i>.</p>
Command Line Redirection	You can redirect the standard streams <code>stdin</code> , <code>stderr</code> , and <code>stdout</code> from the command line or when calling programs using the <code>system()</code> function.
National Language Support	z/OS C/C++ provides message text in either American English or Japanese. You can dynamically switch between the two languages.
Locale Definition Support	z/OS C/C++ provides a locale definition utility that supports the creation of separate files of internationalization data, or locales. Locales can be used at run time to customize the behavior of an application to national language, culture, and coded character set (code page) requirements. Locale-sensitive library functions, such as <code>isdigit()</code> , use this information.
Coded Character Set (Code page) Support	The z/OS C/C++ compiler can compile C/C++ source written in different EBCDIC code pages. In addition, the <code>iconv</code> utility converts data or source from one code page to another.
Selected Built-in Library Functions	Selected library functions, such as string and character functions, are built into the compiler to improve performance execution. Built-in functions are compiled into the executable, and no calls to the library are generated.
Multithreading	Threads are efficient in applications that allow them to take advantage of any underlying parallelism available in the host environment. This underlying parallelism in the host can be exploited either by forking a process and creating a new address space, or by using multiple threads within a single process. For more information, refer to <i>z/OS C/C++ Programming Guide</i> .

Feature	Description
Multitasking Facility (MTF)	<p>Multitasking is a mode of operation where your program performs two or more tasks at the same time. z/OS C provides a set of library functions that perform multitasking. These functions are known as the Multitasking Facility (MTF). MTF uses the multitasking capabilities of z/OS to allow a single z/OS C application program to use more than one processor of a multiprocessing system simultaneously.</p> <p>Note: XPLINK is not supported in an MTF environment. You can also use threads to perform multitasking with or without XPLINK, as described in <i>z/OS C/C++ Programming Guide</i>.</p>
Packed Structures and Unions	z/OS C provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of an z/OS C program or to define structures that are laid out according to COBOL or PL/I structure layout rules.
Fixed-Point (Packed) Decimal Data	<p>z/OS C supports fixed-point (packed) decimal as a native data type for use in business applications. The packed data type is similar to the COBOL data type COMP-3 or the PL/I data type FIXED DEC, with up to 31 digits of precision.</p> <p>The Application Support Class Library provides the Binary Coded Decimal Class for C++ programs.</p>
Long Name Support	For portability, external names can be mixed case and up to 1024 characters in length. For C++, the limit applies to the mangled version of the name.
Large File Support	Enables you to use hierarchical file system (HFS) files that are larger than 2 gigabytes.
System Calls	You can call commands or executable modules using the <code>system()</code> function under z/OS, z/OS UNIX, and TSO. You can also use the <code>system()</code> function to call EXECs on z/OS and TSO, or Shell scripts using z/OS UNIX.
Exploitation of ESA	Support for z/OS, IMS/ESA, Hiperspace expanded storage, and CICS/ESA allows you to exploit the features of the ESA.
Exploitation of hardware	<p>Use the ARCHITECTURE compiler option to select the minimum level of machine architecture on which your program will run. ARCH(2) instructs the compiler to generate faster instruction sequences available only on newer machines. ARCH(3) also generates these faster instruction sequences and enables support for IEEE 754 Binary Floating-Point instructions. Code compiled with ARCH(2) runs on a G2, G3, G4, and 2003 processor and code compiled with ARCH(3) runs on a G5 or G6 processor, and follow-on models.</p> <p>Use the TUNE compiler option to optimize your application for a selected machine architecture. TUNE impacts performance only; it does not impact the processor model on which you will be able to run your application. TUNE(3) optimizes your application for the newer G4, G5, and G6 processors. TUNE(2) optimizes your application for other architectures. For information on which machines and architectures support the above options, refer to the ARCHITECTURE and TUNE compiler information in <i>z/OS C/C++ User's Guide</i>.</p>
Built-in Functions	Use built-in functions to utilize specific hardware instructions that are otherwise inaccessible to C/C++ programs. For information on using built-in functions, see the appendix on built-in functions in <i>z/OS C/C++ User's Guide</i> .

Related Publications

The following titles are related to the z/OS C/C++ product.

- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C Curses*, SA22-7820
- *z/OS C/C++ Compiler and Run-Time Migration Guide*, GC09-4913

- *z/OS C/C++ Messages Guide*, GC09-4819
- *IBM Open Class Library User's Guide*, SC09-4811
- *IBM Open Class Library Reference*, SC09-4812
- *Debug Tool User's Guide and Reference*, SC09-2137

Softcopy Books

All of the z/OS C/C++ publications are supplied in PDF format. They are also available at the following Web Site:

<http://www.ibm.com/software/ad/c390/czos/czosdocs.html>

To read a PDF file, use the Adobe Acrobat Reader, which can be downloaded for free from the Adobe Web Site:

<http://www.adobe.com>

z/OS C/C++ on the World Wide Web

Additional information on z/OS C/C++ is available on the World Wide Web on the z/OS C/C++ home page at:

<http://www.ibm.com/software/ad/c390/czos>





This page contains late-breaking information about the z/OS C/C++ product, including the compiler, the class libraries, and utilities. It also contains a tutorial on the source level interactive debugger. There are links to other useful information, such as the z/OS C/C++ information library and the libraries of other z/OS elements that are available on the Web. The z/OS C/C++ home page also contains samples that you can download, and links to other related Web sites.

Chapter 1. Scope and Linkage

Scope is the mechanism by which it is possible to limit the visibility of declarations in a program. Broadly speaking, it is the general context that differentiates meanings of entity names.

Scope

The area of the code where an identifier is visible is referred to as the *scope* of the identifier. The following are the kinds of scopes:


- Local
- Function
- Function prototype
-  Global or  Global namespace
-  Namespace
-  Class


The scope of a name is determined by the location of the name's declaration.

In all declarations the identifier is in scope before the initializer. The following example demonstrates this:

```
int x;  
void f() {  
  
    int x = x;  
}
```

The `x` declared in function `f()` has local scope, not global namespace scope.

 A function name that is first declared as a friend of a class is in the innermost nonclass scope that encloses the class. If the friend function is a member of another class, it has the scope of that class. The scope of a class name first declared as a friend of a class is the first nonclass enclosing scope.

 The implicit declaration of the class is not visible until another declaration of that same class is seen.

RELATED REFERENCES

- "Local Scope"
- "Function Scope" on page 2
- "Function Prototype Scope" on page 2
- "Global Scope" on page 2
- "Chapter 10. Namespaces" on page 261
- "Class Scope" on page 3

Local Scope

A name has *local scope* or *block scope* if it is declared in a block. A name with local scope can be used in that block and in blocks enclosed within that block, but the name must be declared before it is used. When the block is exited, the names declared in the block are no longer available.

Scope

Parameter names for a function have the scope of the outermost block of that function. Also if the function is declared and not defined, these parameter names have function prototype scope.

► **C++** If a local variable is a class object with a destructor, the destructor is called when control passes out of the block in which the class object was constructed.

When one block is nested inside another, the variables from the outer block are usually visible in the nested block. However, if the declaration of a variable in a nested block has the same name as a variable that is declared in an enclosing block, the declaration in the nested block hides the variable that was declared in the enclosing block. The original declaration is restored when program control returns to the outer block. This is called *block visibility*.

RELATED REFERENCES

- “Block Statement” on page 179
- “Function Prototype Scope”
- “Destructors” on page 350

Function Scope

The only type of identifier with *function scope* is a label name. A label is implicitly declared by its appearance in the program text and is visible throughout the function that declares it.

A label can be used in a goto statement before the actual label is seen.

RELATED REFERENCES

- “Labels” on page 177

Function Prototype Scope

In a function declaration (also called a *function prototype*) or in any function declarator — except the declarator of a function definition — parameter names have *function prototype scope*. Function prototype scope terminates at the end of the nearest enclosing function declarator.

RELATED REFERENCES

- “Function Declarations” on page 154

Global Scope

► **C** A name has *global scope* if the identifier’s declaration appears outside of any block. A name with global scope and internal linkage is visible from the point where it is declared to the end of the translation unit. (A *translation unit* is a source code file after preprocessing with include files.)

► **C++** A name has *global namespace scope* if the identifier’s declaration appears outside of all blocks and classes. A name with global namespace scope and internal linkage is visible from the point where it is declared to the end of the translation unit.

A name with global (namespace) scope is also accessible for the initialization of global variables. If that name is declared **extern**, it is also visible at link time in all object files being linked.

RELATED REFERENCES

- “Chapter 10. Namespaces” on page 261
- “Internal Linkage” on page 5
- “extern Storage Class Specifier” on page 37

Class Scope

C++ A name declared within a member function hides a declaration of the same name whose scope extends to or past the end of the member function’s class.

The scope of a declaration that extends to or past the end of a class definition also extends to the regions defined by its member definitions and any portion of the declarator part of such definitions which follows the identifier.

The name of a class member has *class scope* and can only be used in the following cases:

- In a member function of that class
- In a member function of a class derived from that class
- After the . (dot) operator applied to an instance of that class
- After the . (dot) operator applied to an instance of a class derived from that class, as long as the derived class does not hide the name
- After the -> (arrow) operator applied to a pointer to an instance of that class
- After the -> (arrow) operator applied to a pointer to an instance of a class derived from that class, as long as the derived class does not hide the name
- After the :: (scope resolution) operator applied to the name of a class
- After the :: (scope resolution) operator applied to a class derived from that class.

RELATED REFERENCES

- “Chapter 12. Classes” on page 283
- “Member Functions” on page 295
- “Derivation” on page 317
- “Dot Operator .” on page 107
- “Arrow Operator ->” on page 107
- “C++ Scope Resolution Operator ::” on page 102
- “Scope of Class Names” on page 287

Example of Scope in C

The following example declares the variable *x* on line 1, which is different from the *x* it declares on line 2. The declared variable on line 2 has function prototype scope and is visible only up to the closing parenthesis of the prototype declaration. The variable *x* declared on line 1 resumes visibility after the end of the prototype declaration.

```

1  int x = 4;                /* variable x defined with file scope */
2  long myfunc(int x, long y); /* variable x has function      */
3                                /* prototype scope          */
4  int main(void)
5  {
6      /* . . . */
7  }
```

Scope

The following program illustrates blocks, nesting, and scope. The example shows two kinds of scope: file and block. The `main()` function prints the values 1, 2, 3, 0, 3, 2, 1 on separate lines. Each instance of `i` represents a different variable.

```
#include <stdio.h>
int i = 1;                                /* i defined at file scope */

int main(int argc, char * argv[])
{
    printf("%d\n", i);                    /* Prints 1 */
    {
        int i = 2, j = 3;                /* i and j defined at
                                           block scope */
        printf("%d\n%d\n", i, j);        /* Prints 2, 3 */
        {
            int i = 0;                   /* i is redefined in a nested block
                                           /* previous definitions of i are hidden */
            printf("%d\n%d\n", i, j);    /* Prints 0, 3 */
        }
        printf("%d\n", i);                /* Prints 2 */
    }
    printf("%d\n", i);                    /* Prints 1 */
    return 0;
}
```

Name Hiding

C++ If a class name or enumeration name is in scope and not hidden it is *visible*. A class name or enumeration name can be hidden by an explicit declaration of that same name — as an object, function, or enumerator — in a nested declarative region or derived class. The class name or enumeration name is hidden wherever the object, function, or enumerator name is visible. This process is referred to as *name hiding*.

In a member function definition, the declaration of a local name hides the declaration of a member of the class with the same name. The declaration of a member in a derived class hides the declaration of a member of a base class of the same name.

Suppose a name `x` is a member of namespace `A`, and suppose that the members of namespace `A` are visible in a namespace `B` because of a `using` declaration. A declaration of an object named `x` in namespace `B` will hide `A::x`. The following example demonstrates this:

```
#include <iostream>
#include <typeinfo>
using namespace std;

namespace A {
    char x;
};

namespace B {
    using namespace A;
    int x;
```

```
};

int main() {
    cout << typeid(B::x).name() << endl;
}
```

The following is the output of the above example:

```
int
```

The declaration of the integer `x` in namespace `B` hides the character `x` introduced by the using declaration.

RELATED REFERENCES

- “Chapter 12. Classes” on page 283
- “Member Functions” on page 295
- “Member Scope” on page 297
- “Chapter 10. Namespaces” on page 261
- “Using Directive” on page 266

Program Linkage

Linkage determines whether identifiers that have identical names refer to the same object, function, or other entity, even if those identifiers appear in different translation units. (A *translation unit* is a source code file after preprocessing with include files.) The linkage of an identifier depends on how it was declared. There are three kinds of linkage:

- If an identifier `x` has *internal linkage*, every appearance of `x` within one translation unit refers to the same entity.
- If an identifier `x` has *external linkage*, every appearance of `x` across any translation unit (of the same program) refers to the same entity.
- If an identifier `x` has *no linkage*, every appearance of `x` refers to a unique entity.

RELATED REFERENCES

- “Internal Linkage”
- “External Linkage” on page 6
- “No Linkage” on page 7

Internal Linkage

The following kinds of identifiers have internal linkage:

- Objects, references, functions or `C++` function templates explicitly declared **static**.
- Objects or references declared in namespace scope (or global scope in C) with the specifier **const** and neither explicitly declared **extern**, nor previously declared to have external linkage.
- Data members of an anonymous union.
- `C++` Identifiers declared in the unnamed namespace.

A function declared inside a block will usually have external linkage. An object declared inside a block will usually have external linkage if it is specified **extern**. If a variable that has **static** storage is defined outside a function, the variable has internal linkage and is available from the point where it is defined to the end of the current translation unit.

Program Linkage

A class that has no static members or noninline member functions, and that has not been used in the declaration of an object or function or class is local to its compilation unit.


If the declaration of an identifier has the keyword **extern** and if a previous declaration of the identifier is visible at namespace or global scope, the identifier has the same linkage as the first declaration.

RELATED REFERENCES



- “static Storage Class Specifier” on page 42
- “volatile and const Qualifiers” on page 69
- “extern Storage Class Specifier” on page 37
- “Global Scope” on page 2
- “Anonymous Unions” on page 62
- “Unnamed Namespaces” on page 264

External Linkage

The following kinds of identifiers with namespace scope (or global scope in C) have external linkage:

- An object, reference, or function unless it has internal linkage.
-  A named class or enumeration.
- An enumerator of an enumeration that has external linkage.
- A template, unless it is a function template with internal linkage
- A namespace, unless it is declared in an unnamed namespace

The following also have external linkage:

- Member functions, static data members, classes, or enumerations if the class that they belong to has external linkage.
- Identifiers with  namespace scope or local scope that have the keyword **extern** in their declarations.
-  Static class members and noninline member functions

If a previous declaration of an object or function is visible in an enclosing scope, the identifier has the same linkage as the first declaration. For example, a variable or function that is first declared with the keyword **static** and later declared with the keyword **extern** has internal linkage. However, a variable or function that has no linkage and later declared with a linkage specifier will have the linkage you have specified.

RELATED REFERENCES

- “Scope” on page 1
- “static Storage Class Specifier” on page 42
- “extern Storage Class Specifier” on page 37
- “Chapter 12. Classes” on page 283
- “Enumerations” on page 65
- “typedef” on page 43
- “Chapter 10. Namespaces” on page 261
- “Static Members” on page 303
- “Inline Functions” on page 174

No Linkage

The following kinds of identifiers have no linkage:

- Names that have neither external or internal linkage
- Names declared in local scopes (with exceptions like certain entities declared with the **extern** keyword)
- Identifiers that do not represent an object or a function, including labels, enumerators, **typedef** names that refer to entities with no linkage, type names, function parameters, and template names

You cannot use a name with no linkage to declare an entity with linkage. For example, you cannot use the name of a class or enumeration or a typedef name referring to an entity with no linkage to declare an entity with linkage. The following example demonstrates this:

```
int main() {
    struct A { };
    // extern A a1;
    typedef A myA;
    // extern myA a2;
}
```

The compiler will not allow the declaration of a1 with external linkage. Class A has no linkage. The compiler will not allow the declaration of a2 with external linkage. The typedef name a2 has no linkage because A has no linkage.

RELATED REFERENCES

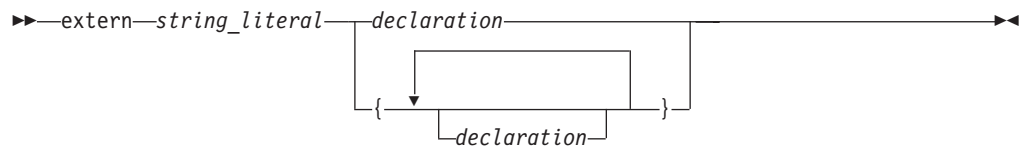
- “Program Linkage” on page 5
- “Scope” on page 1
- “typedef” on page 43
- “static Storage Class Specifier” on page 42
- “extern Storage Class Specifier” on page 37

Linkage Specifications — Linking to Non-C++ Programs

► **C++** You can link C++ object modules to object modules produced using other source languages such as C by using a *linkage specification*.

► **z/OS** In z/OS C, the `#pragma linkage` directive specifies non-C declarations.

The syntax is:



The *string_literal* is used to specify the linkage associated with a particular function.

CCNX02J

// This example illustrates linkage specifications.

```
extern "C" int printf(const char*,...);
```

Linkage Specifications

```
int main(void)
{
    printf("hello\n");
}
```

Here the *string_literal*, "C", tells the compiler that the routine `printf(const char*,...)` is a C library function.

Note: This example is not guaranteed to work on all platforms. The only safe way to declare a function from the C library in a C++ program is to include the appropriate header. In this example you would substitute the line of code with **extern** with the following line:

```
#include <stdio.h>
```

String literals used in linkage specifications should be considered as case-sensitive.

All platforms support the following values for *string_literal*

"C++"	Unless otherwise specified, objects and functions have this default linkage specification.
"C"	Indicates linkage to a C procedure

Name Spaces of Identifiers

Name spaces are the various syntactic contexts in which an identifier can be used. Within the same context and the same scope, an identifier uniquely identifies an entity. Note that the term *name space* as used here does not refer to the C++ namespace language feature. The z/OS compiler sets up *name spaces* to distinguish among identifiers referring to different kinds of entities. Identical identifiers in different name spaces do not interfere with each other, even if they are in the same scope.

You must assign unique names within each name space to avoid conflict. You can use the same identifier to declare different objects as long as each identifier is unique within its name space. The syntactic context of an identifier within a program lets the compiler resolve its name space without ambiguity.

You can redefine identifiers in the same name space but within enclosed program blocks, as described in "Scope" on page 1.

Within each of the following four name spaces, the identifiers must be unique.

- *Tags* of these types must be unique within a single scope:
 - Enumerations
 - Structures and unions
- *Members* of structures, unions, and classes must be unique within a single structure, union, or class type.
- *Statement labels* have function scope and must be unique within a function.
- All other *ordinary identifiers* must be unique within a single scope:
 - C function names (C++ function names can be overloaded)
 - Variable names
 - Names of function parameters
 - Enumeration constants
 - typedef names.

Name Spaces of Identifiers

Structure tags, structure members, variable names, and statement labels are in four different name spaces. No conflict occurs among the four items named `student` in the following example:

```
int get_item()
{
    struct student      /* structure tag      */
    {
        char student[20]; /* structure member */
        int section;
        int id;
    } student;          /* structure variable */

    goto student;
    student::;          /* null statement label */
    return 0;
}
```

z/OS C/C++ interprets each occurrence of `student` by its context in the program. For example, when `student` appears after the keyword `struct`, it is a structure tag. When `student` appears after either of the member selection operators `.` or `->`, the name refers to the structure member. When `student` appears after the `goto` statement, z/OS C/C++ passes control to the null statement label. In other contexts, the identifier `student` refers to the structure variable.

Name Spaces of Identifiers

Chapter 2. Lexical Elements

This section contains discussions of the basic lexical elements and conventions of the C and C++ programming languages: tokens, character sets, comments, identifiers, and literals.

Tokens

Source code is treated during preprocessing and compilation as a sequence of *tokens*. A token is the smallest independent unit of meaning of a program as defined by the compiler. There are five different types of tokens:

- Identifiers
- Keywords
- Literals
- Operators
- Punctuators

Adjacent identifiers, keywords and literals must be separated with white space. Other tokens should be separated by white space to make the source code more readable. White space includes blanks, horizontal and vertical tabs, new lines, form feeds and comments.

RELATED REFERENCES

- “Identifiers” on page 18
- “Keywords” on page 20
- “Literals” on page 23
- “Chapter 5. Expressions and Operators” on page 95
- “Source Program Character Set”

Source Program Character Set

The following lists the basic *source character set* that must be available at both compile and run time:

- The uppercase and lowercase letters of the English alphabet

```
...a b c d e f g h i j k l m n o p q r s t u v w x y z  
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```


- The decimal digits 0 through 9

```
0 1 2 3 4 5 6 7 8 9
```

- The following punctuators (A *punctuator* is a character that has syntactic and semantic meaning, but does not specify an operation that produces a value. Depending on the context, a punctuator can also be an operator.):

```
! " # % & ' ( ) * + , - . / :  
; < = > ? [ \ ] _ { }
```

- The caret (^) character in ASCII (bitwise exclusive OR symbol) or the equivalent not (~) character in EBCDIC
- The split vertical bar (|) character in ASCII, which may be represented by the vertical bar (|) character on EBCDIC systems
- The space character
- The control characters representing new-line, horizontal tab, vertical tab, and form feed, and end of string (NULL character)

 The remainder of this section discusses points specific to the z/OS C/C++ implementation.

Character Set

z/OS C/C++ uses the number sign (#) character for preprocessing only, and treats the _ (underscore) character as a normal letter.

The execution character set also includes control characters that represent alert, backspace, carriage return, and new-line.

In a source file, a record contains one line of source text; the end of a record indicates the end of a source line.

The encoding of the following characters from the basic character set may vary between the source-code generation environment and the run-time environment:

! # ' [] \ { } ~ ^ |

The z/OS C/C++ compiler normalizes the encoding of source files indicated by the #pragma filetag directive and the LOCALE compile time option to the encoding defined by code page 1047.

The compiler uses the character set that is specified for the LOCALE option for any output. This includes:

- Listings that contain identifier names and source code
- String literals and character constants that are emitted in the object code
- Messages generated by the compiler

However, this does not include the source-code annotation in the pseudo-assembly listings.

Depending on the EBCDIC encoding that your installation uses, you can express the ^ and | characters as ~ and | respectively. This book refers to the ^ and | symbols as the *caret* and *vertical bar*, respectively. If you do not specify the NOLOCALE compile-time option, z/OS C/C++ does not perform normalization. It assumes that the character set encoding is the IBM-1047 code page. In this case, it recognizes both the broken and unbroken vertical bars as the vertical bar. The caret and logical not sign are recognized as the caret. For a detailed description of the #pragma filetag directive and the LOCALE option, refer to the description of internationalization, locales, and character sets in the *z/OS C/C++ Programming Guide*.

The compiler recognizes and supports the additional characters (the extended character set) which you can meaningfully use in string literals and character constants. The support for extended characters includes the multibyte character sets.

z/OS systems represent multibyte characters by using Shiftout <S0> and Shiftin <SI> pairs. Strings are of the form:

<S0> x y z <SI>

Or they can be mixed:

<S0> x <SI> y z
x <S0> y <SI> z

In the above, two bytes represent each character between the <S0> and <SI> pairs. z/OS C/C++ restricts multibyte characters to character constants, string constants, and comments.

Refer to the *z/OS C/C++ Run-Time Library Reference* for a discussion on strings that are passed to library routines, and to “Character Literals” on page 28 of this

book for information on character constants. If you specify a lowercase a as part of an identifier name, you cannot substitute an uppercase A in its place. You must use the lowercase letter.

RELATED REFERENCES

- “Chapter 5. Expressions and Operators” on page 95

Escape Sequences

You can represent any member of the execution character set by an *escape sequence*. They are primarily used to put nonprintable characters in character and string literals. For example, you can use escape sequences to put such characters as tab, carriage return, and backspace into an output stream.



An escape sequence contains a backslash (\) symbol followed by one of the escape sequence characters or an octal or hexadecimal number. A hexadecimal escape sequence contains an x followed by one or more hexadecimal digits (0-9, A-F, a-f). An octal escape sequence uses up to three octal digits (0-7). The value of the hexadecimal or octal number specifies the value of the desired character or wide character.

Note: The line continuation sequence (\ followed by a new-line character) is not an escape sequence. It is used in character strings to indicate that the current line continues on the next line.

The escape sequences and the characters they represent are:

Escape Sequence	Character Represented
\a	Alert (bell, alarm)
\b	Backspace
\f	Form feed (new page)
\n	New-line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\\	Backslash

The value of an escape sequence represents the member of the character set used at run time. Escape sequences are translated during preprocessing. For example, on a system using the ASCII character codes, the value of the escape sequence \x56 is the letter V. On a system using EBCDIC character codes, the value of the escape sequence \xE5 is the letter V.

Use escape sequences only in character constants or in string literals. An error message is issued if an escape sequence is not recognized.

Character Set

In string and character sequences, when you want the backslash to represent itself (rather than the beginning of an escape sequence), you must use a `\\` backslash escape sequence. For example:

```
cout << "The escape sequence \\n." << endl;
```

This statement results in the following output:

The escape sequence `\\n`.

The Unicode Standard

C++ The Unicode Standard is currently supported only for z/OS C++.

The *Unicode Standard* is a standardized character code designed to encode international texts for display and storage. It uses a unique 16-bit value to represent each individual character. The Unicode standard includes the following:

- Alphabets used in Europe, Africa, and Asia
- Standard characters from China, Japan, Korea, and Taiwan
- Mathematical operators
- Technical symbols

The following diagram illustrates how Unicode assigns a unique 16-bit value to each character:

BIG5	Unicode
陳 0xAFB3	0x9673 陳
煒 0x6DDE	0x7152 煒
鈞 0x76B6	0x921E 鈞
	⋮
Shift-JIS	0xFF73 ウ
ウ 0xB3	0xFF6F ツ
ツ 0xAF	0xFF76 カ
カ 0xB6	

0xAFB3 represents the 陳 character in BIG5 and the characters ウツ in Shift-JIS. Unicode assigns each character with a unique code point. In this case 陳 is assigned with 0x9673 and ウツ with 0xFF73 and 0xFF6F.

Although the 16-bit architecture of Unicode can handle more than 65,000 different characters, the Unicode Standard can extend to handle an additional one million characters by the *surrogate extension* mechanism. This mechanism uses two 16-bit values to represent one character. The Unicode Standard has not used any of these surrogates. (The current standard contains 38,885 characters.)

The Unicode Standard lets you dynamically compose accented characters. In the Unicode Standard, a character and an accent are separate characters. In other character encodings such as ASCII, you select from a set of accented characters.

The Unicode Standard supports bidirectional ordering of languages. Bidirectional language ordering occurs when a script uses two or more languages with different dominant directions. For example, a script would have bidirectional language ordering if it mixes Arabic (which reads from right-to-left) with Greek (which reads from left-to-right). The Unicode Standard includes characters that specify a change of direction.

You can represent Unicode characters in your program by using one of the two following forms, where each x is a hexadecimal digit:

```
\uxxxx
\Uxxxxxxxx
```

The first form, `\uxxxx`, represents a Unicode character that uses one 16-bit value. The second form, `\Uxxxxxxxx`, represents a character that uses two 16-bit values.

Trigraph Sequences

Some characters from the C and C++ character set are not available in all environments. You can enter these characters into a C source program using a sequence of three characters called a *trigraph*. The trigraph sequences are:

??=	#	pound sign
??{	[left bracket
??}]	right bracket
??<	{	left brace
??>	}	right brace
??/	\	backslash
??'	^	caret
??!		vertical bar
??-	~	tilde

The preprocessor replaces trigraph sequences with the corresponding single-character representation.

z/OS In the z/OS C/C++ implementation, the preprocessor makes this replacement by using the code page that is indicated by the `LOCALE` option. If you do not specify the `LOCALE` option, the preprocessor uses code page 1047.

At compile time, the compiler translates the trigraphs found in string literals and character constants into the appropriate characters they represent. These characters are in the coded character set you select by using the `LOCALE` compiler option.

z/OS The z/OS C/C++ compiler will compile source files that were edited using different encoding of character sets. However, they might not compile cleanly. z/OS C/C++ does not compile source files that you edit with the following:

- A character set that does not support all the characters that are specified above, even if the compiler can access those characters by a trigraph.
- A character set for which no one-to-one mapping exists between it and the character set above.

Note: The exclamation mark (!) is a variant character. Its recognition depends on whether or not the `LOCALE` option is active. For more information on variant characters, refer to the *z/OS C/C++ Programming Guide*.

Character Set

Example

```
some_array??(i??) = n;
```

Represents:

```
some_array[i] = n;
```

Digraph Characters

z/OS You can represent unavailable characters in a z/OS C or C++ source program by using a combination of two keystrokes that are called a *digraph character*. The preprocessor reads digraphs as tokens during the preprocessor phase.

The digraph characters are:

%: or %%	#	number sign
<:	[left bracket
:>]	right bracket
<%	{	left brace
%>	}	right brace
%: or %:	##	preprocessor macro concatenation operator
%%%		

You can create digraphs by using macro concatenation. z/OS C/C++ does not replace digraphs in string literals or in character literals. For example:

```
char *s = "<%%>"; // stays "<%%>"

switch (c)
{
    case '<%': { /* ... */ } // stays '<% '
    case '%>': { /* ... */ } // stays '%> '
}
```

The NODIGRAPH option disables processing of digraphs. The NODIGRAPH option is on by default.

Comments

A *comment* is text replaced during preprocessing by a single space character; the compiler therefore ignores all comments.

There are two kinds of comments:

- **C** The /* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the */ characters. This kind of comment is commonly called a *C-style comment*.
- **C++** The // (two slashes) characters followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. This kind of comment is commonly called a *single-line comment* or a *C++ comment*. A C++ comment can span more than one physical source line if it is joined into one logical source line with line-continuation (\) characters. The backslash character can also be represented by a trigraph.
- **z/OS** If the SSCOMM compiler option is in effect when you compile a C program, double slashes (//) also specify the beginning of a comment. C++ permits double-slash comments as part of the language definition.

You can put comments anywhere the language allows white space. You cannot nest C-style comments inside other C-style comments.

Multibyte characters can also be included within a comment.

Note: The `/*` or `*/` characters found in a character constant or string literal do not start or end comments.

In the following program, the second `printf()` is a comment:

```
#include <stdio.h>

int main(void)
{
    printf("This program has a comment.\n");
    /* printf("This is a comment line and will not print.\n"); */
    return 0;
}
```

Because the second `printf()` is equivalent to a space, the output of this program is:

This program has a comment.

Because the comment delimiters are inside a string literal, `printf()` in the following program is not a comment.

```
#include <stdio.h>

int main(void)
{
    printf("This program does not have \
/* NOT A COMMENT */ a comment.\n");
    return 0;
}
```

The output of the program is:

This program does not have
/* NOT A COMMENT */ a comment.

You cannot nest C-style comments. Each comment ends at the first occurrence of `*/`.

In the following example, the comments are highlighted:

/* A program with nested comments. */

```
#include <stdio.h>

int main(void)
{
    test_function();
    return 0;
}

int test_function(void)
{
    int number;
    char letter;
    /*
number = 55;
letter = 'A';
*/
}
```

Comments

```
/* number = 44; */
*/
return 999;
}
```

In `test_function`, the compiler reads the first `/*` through to the first `*/`. The second `*/` causes an error. To avoid commenting over comments already in the source code, you should use conditional compilation preprocessor directives to cause the compiler to bypass sections of a program. For example, instead of commenting out the above statements, change the source code in the following way:

```
/* A program with conditional compilation to avoid nested comments.
*/
#define TEST_FUNCTION 0
#include <stdio.h>


int main(void)
{
    test_function();
    return 0;
}

int test_function(void)
{
    int number;
    char letter;
    #if TEST_FUNCTION
        number = 55;
        letter = 'A';
        /*number = 44;*/
    #endif /*TEST_FUNCTION */
}
```

You can nest single line comments within C-style comments. For example, the following program will not output anything:

```
#include <stdio.h>

int main(void)
{
    /*
    printf("This line will not print.\n");
    // This is a single line comment
    // This is another single line comment
    printf("This line will also not print.\n");
    */
    return 0;
}
```

 “Chapter 9. Preprocessor Directives” on page 195 describes conditional compilation preprocessor directives. You can include multibyte characters with a comment.

RELATED REFERENCES

- “Trigraph Sequences” on page 15

Identifiers

Identifiers consist of an arbitrary number of letters or digits. They provide names for the following language elements:

- Functions
- Objects
- Labels

- Function parameters
- Macros and macro parameters
- Typedefs
- Enumerated types and enumerators
- **z/OS C++** Classes and class members
- **z/OS C++** Templates
- **z/OS C++** Template parameters
- **z/OS C++** Namespaces
- Struct and union names

An identifier has the form:



Case Sensitivity and Special Characters in Identifiers

The compiler distinguishes between uppercase and lowercase letters in identifiers. For example, PROFIT and profit represent different identifiers.

Note: **z/OS** If you do not use the z/OS C compiler long name support and if the names have external linkage, STOCKONHOLD and stockonhold, for example, both refer to the same object; all external names are truncated to eight characters and uppercased in the object file. For information on long name support, see “longname” on page 241. For more information on the binder and the prelinker, see *z/OS C/C++ User's Guide*.

Avoid creating identifiers that begin with an underscore (`_`) for function names and variable names.

The first character in an identifier must be a letter. The `_` (underscore) character is considered a letter; however, identifiers beginning with an underscore are reserved by the compiler for identifiers at global namespace scope.

Identifiers that contain two consecutive underscores or begin with an underscore followed by a capital letter are reserved in all contexts.

z/OS Although the names of system calls and library functions are not reserved words if you do not include the appropriate headers, avoid using them as identifiers. Duplication of a predefined name can lead to confusion for the maintainers of your code and can cause errors at link time or run time. If you include a library in a program, be aware of the function names in that library to avoid name duplications. You should always include the appropriate headers when using standard library functions.

Significant Characters in Identifiers

z/OS In general, z/OS C/C++ truncates external and internal identifiers after 1024 characters. However, the C compiler truncates external identifiers after 8 characters

Identifiers

if the `NOLONGNAME` compile-time option is in effect. Also, the C++ compiler truncates external identifiers that do not have C++ linkage after 8 characters if the `NOLONGNAME` compile-time option is in effect.

Keywords

Keywords are identifiers reserved by the language for special use. Although you can use them for preprocessor macro names, it is poor programming style. Only the exact spelling of keywords is reserved. For example, `auto` is reserved but `AUTO` is not. The following lists the keywords common to both the C and C++ languages:

auto	double	long	typedef
break	else	register	union
case	enum	return	unsigned
char	extern	short	void
const	float	signed	volatile
continue	for	sizeof	while
default	goto	static	
do	if	struct	
	int	switch	

► **C++** The C++ language also reserves the following keywords:

asm	export	private	true
bool	false	protected	try
catch	friend	public	typeid
class	inline	reinterpret_cast	typename
const_cast	mutable	static_cast	using
delete	namespace	template	virtual
dynamic_cast	new	this	wchar_t
explicit	operator	throw	

Keywords for language extensions

► **z/OS** In addition to language keywords, z/OS C/C++ reserves identifiers for use in language extensions and for future use. For this usage, the following keyword is common to both the z/OS C and C++ languages:

__callback

► **C** z/OS C reserves the following for use in language extensions:

_Packed **__packed**

► **C++** z/OS C++ reserves the following for use in language extensions:

_cdecl **_Export**

Alternative representations of operators and punctuators

► **C++** In addition to the reserved language and language extension keywords, the following alternative representations of operators and punctuators are also reserved in C++ when the `DIGRAPH` option is specified:

and	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

Alternative Tokens

z/OS C and C++ provide alternative representations for some operators and punctuators. However, there are slight differences between the two treatments: some alternative representations for C++ are treated as macros in C.

The following table lists the operators and punctuators and their alternative representation:

Table 1. Digraphs that apply to both C and C++

Operator or Punctuator	Alternative Representation
{	<%
}	%>
[<:
]	:>
#	%:
##	%.%:

C++ In addition to operators and punctuators listed above, C++ provides the following alternative representations:

Table 2. Digraphs for C++ only

Operator or Punctuator	Alternative Representation
&&	and
	bitor
	or
^	xor
~	compl
&	bitand
&=	and_eq
=	or_eq
^=	xor_eq
!	not
!=	not_eq

C The alternative representations in the preceding table are not provided in C, but are defined as macros in the system header file `iso646.h`.

z/OS C/C++ External Name Mapping

z/OS z/OS C/C++ maps the names of variables or functions that have external linkages to names that are used in the object module. Note that this mapping is only done when the `NOLONGNAME` compiler option is in effect.

When you compile a program, refer to the following guidance for using names of variables or functions with external linkage:

- Do not use names of the library functions for user-defined functions.
- Some functions in the C library and C run-time environment begin with two underscores (`_ _`). Do not use an underscore as the first letter of an identifier.
- The compiler maps each underscore to an at sign (`@`) for external names without C++ linkage, except when you compile a program with the `LONGNAME` compile-time option. In that case, the underscore remains as an underscore.
- IBM-provided functions have names that begin with `IBM`, `CEE`, and `PLI`. Avoid using these names as the z/OS C/C++ compiler changes these names to prevent conflicts between run-time functions and user-defined names. It changes all static or extern variable names that begin with `IBM`, `CEE`, and `PLI` in your source program to `IB$`, `CE$`, and `PL$`, respectively, in the object module. If you are using interlanguage calls, avoid using these prefixes. The compiler of the calling or called language may or may not change these prefixes in the same manner as the z/OS C/C++ compiler does. All of this is completely integrated into the z/OS C/C++ compiler, Debug Tool, and LE/370.

To call an external program or access an external variable that begins with `IBM`, `CEE`, and `PLI`, use the `#pragma map` preprocessor directive. The following is an example of `#pragma map` that forces an external name to be `IBMENTRY`.

```
#pragma
map(ibmentry,"IBMENTRY")
```

For more information on the `#pragma map` directive, see “map” on page 242.

z/OS Long Name Support

z/OS If you do not specify the `LONGNAME` option when you compile your code with the C compiler, the compiler maps an underscore to an at sign. It also truncates external names to 8 characters and changes them to uppercase. The C++ compiler makes the same changes to external identifiers that do not have C++ linkage if you do not specify the `LONGNAME` option.

For example, consider if you compile the following C program and do not specify the `LONGNAME` option:

```
int test_name[4] = { 4, 8, 9, 10 };
int test_namesum;

int main(void) {
    int i;
    test_namesum = 0;

    for (i = 0; i < 4; i++)
        test_namesum += test_name[i];
    printf("sum is %d\n", test_namesum);
}
```

In the above example, the C compiler displays the following message:

```
ERROR CCN3244 ./sum.c:2 External name TEST_NAM cannot be redefined.
```


The compiler changes the external names `test_namesum` and `test_name` to uppercase and truncates them to 8 characters. If you specify the `CHECKOUT` compile-time option, the compiler will generate two informational messages to this effect. Because the truncated names are now the same, the compiler produces an error message and terminates the compilation.

If you compile the previous program with the `LONGNAME` compile-time option, the compiler does not produce any warning or error messages. However, if you specify the `LONGNAME` option, you must bind your program with the binder to produce a program object in a PDSE. Otherwise you must use the prelinker.

The `LONGNAME` compile-time option supports mixed case, external names of up to 1024 characters for z/OS C/C++ programs.

Object modules that are produced by compiling with `LONGNAME` have external names that are mixed case and up to 1024 characters long. Object modules that are produced by compiling with `NOLONGNAME` have uppercase external names that are limited to a length of 8 characters.


To use external C names that are longer than 8 characters or external C++ names without C++ linkage that are longer than 8 characters, you can, in your source code:

- Use the `#pragma map` directive to map long external names in the source code to 8 or less characters in the object module.

```
#pragma map(verylongname,"sname")
```
- Use the long name support that is provided by the compile-time option `LONGNAME`. To use the long name support, you must do the following:
 - Use the `LONGNAME` compile-time option when compiling your program.
 - Use the binder to produce a program object in a PDSE, or use the prelinker. For more information on the binder, prelinker, and `LONGNAME` compile-time option, see the *z/OS C/C++ User's Guide*.

Literals

A *literal* does not change its value while the program is running. The value of any literal must be in the range of representable values for its type. The following are the available types of literals:

- Integer
- Character
- Floating-point
- Fixed-Point Decimal Constants (z/OS C)
- String
-  Boolean

 The C language uses the term *constant* in place of the term *literals*.

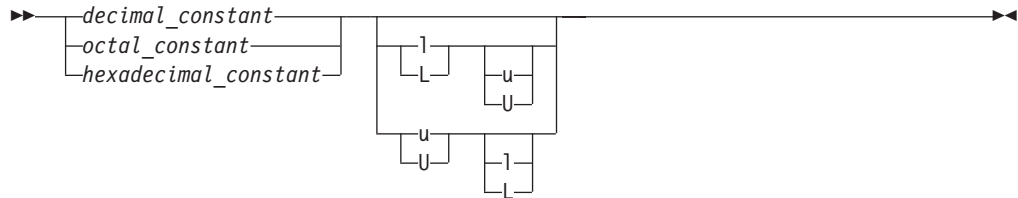
RELATED REFERENCES

- “Integer Literals” on page 24
- “Floating-Point Literals” on page 25
- “Character Literals” on page 28
- “String Literals” on page 29
- “Boolean Literals” on page 31

Literals

Integer Literals

Integer literals can represent decimal, octal, or hexadecimal values. They are numbers that do not have a decimal point or an exponential part. However, an integer literal may have a prefix that specifies its base, or a suffix that specifies its type.



z/OS An integer constant without a suffix cannot have a value greater than `ULONG_MAX`. An integer constant with a suffix that contains `LL` cannot have a value greater than `ULLONG_MAX`. In these cases, the compiler will issue an *out of range* error message. For information on the `ULONG_MAX` and the `ULLONG_MAX` macros, see the *z/OS C/C++ Run-Time Library Reference*.

The data type of an integer literal is determined by its form, value, and suffix. The following table lists the integer literals and shows the possible data types. The smallest data type that can represent the constant value is used to store the constant.

Integer Literal	Possible Data Types
unsuffixed decimal	int, long int, unsigned long int
unsuffixed octal	int, unsigned int, long int, unsigned long int
unsuffixed hexadecimal	int, unsigned int, long int, unsigned long int
suffixed by u or U	unsigned int, unsigned long int
suffixed by l or L	long int, unsigned long int
suffixed by both u or U , and l or L	unsigned long int
suffixed by ll or LL	long long int, unsigned long long int (not z/OS)
suffixed by both u or U , and ll or LL	unsigned long long int

A plus (+) or minus (-) symbol can precede an integer literal. The operator is treated as a unary operator rather than as part of the literal.

RELATED REFERENCES

- “Decimal Integer Literals”
- “Hexadecimal Decimal Literals” on page 25
- “Octal Decimal Literals” on page 25
- “Integer Variables” on page 49

Decimal Integer Literals

A *decimal integer literal* contains any of the digits 0 through 9. The first digit cannot be 0.



Integer literals beginning with the digit 0 are interpreted as an octal integer literal rather than as a decimal integer literal.

The following are examples of decimal literals:

```
485976
-433132211
+20
5
```

A plus (+) or minus (-) symbol can precede the decimal integer literal. The operator is treated as a unary operator rather than as part of the literal.

Hexadecimal Integer Literals

A *hexadecimal integer literal* begins with the 0 digit followed by either an x or X, followed by any combination of the digits 0 through 9 and the letters a through f or A through F. The letters A (or a) through F (or f) represent the values 10 through 15, respectively.



The following are examples of hexadecimal integer literals:

```
0x3b24
0XF96
0x21
0x3AA
0X29b
0X4bD
```

Octal Integer Literals

An *octal integer literal* begins with the digit 0 and contains any of the digits 0 through 7.



The following are examples of octal integer literals:

```
0
0125
034673
03245
```

Floating-Point Literals

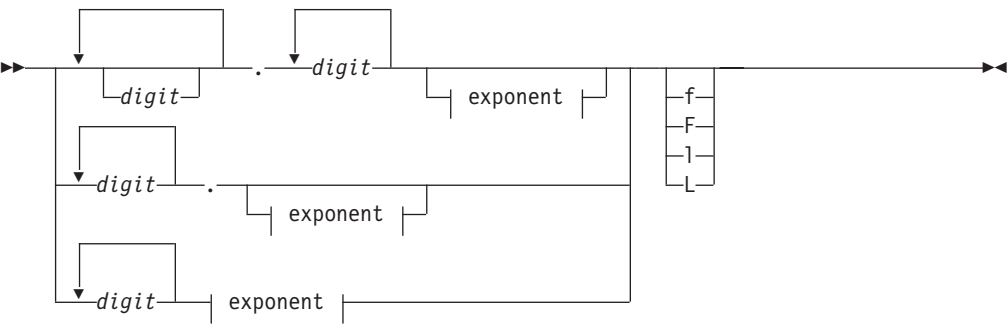
A *floating-point literal* consists of the following:

- an integral part

Literals

- a decimal point
- a fractional part
- an exponent part
- an optional suffix

Both the integral and fractional parts are made up of decimal digits. You can omit either the integral part or the fractional part, but not both. You can omit either the decimal point or the exponent part, but not both.



Exponent:



The magnitude range of **float** is approximately 1.2e-38 to 3.4e38. The magnitude range of **double** or **long double** is approximately 2.2e-308 to 1.8e308. If a floating-point constant is too large or too small, the result is undefined by the language. The ranges depend on which floating point is used: HEX or IEEE.

The suffix **f** or **F** indicates a type of **float**, and the suffix **l** or **L** indicates a type of **long double**. If a suffix is not specified, the floating-point constant has a type **double**.

A plus (+) or minus (-) symbol can precede a floating-point literal. However, it is not part of the literal; it is interpreted as a unary operator.

The following are examples of floating-point literals:

Floating-Point Constant	Value
5.3876e4	53,876
4e-11	0.000000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

► **C** When you use the **printf** function to display a floating-point constant value, make certain that the **printf** conversion code modifiers that you specify are large enough for the floating-point constant value.

RELATED REFERENCES

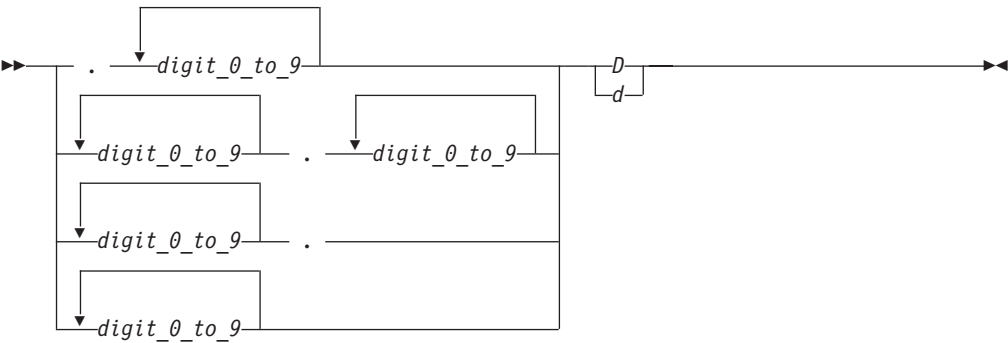
- “Floating-Point Variables” on page 47
- “Unary Expressions” on page 113

Fixed-Point Decimal Constants (z/OS C Only)

► **z/OS** *Fixed-point decimal constants* are an IBM extension to ISO C. This type is available when you specify the **LANGLVL(EXTENDED)** compile-time option.

A fixed-point decimal constant has a numeric part and a suffix that specifies its type. The numeric part can include a digit sequence that represents the whole-number part, followed by a decimal point (.), followed by a digit sequence that represents the fraction part. Either the integral part or the fractional part, or both must be present.

A fixed-point constant has the form:



A fixed-point constant has two attributes:

- Number of digits (size)
- Number of decimal places (precision).

The suffix **D** or **d** indicates a fixed-point constant.

The following are examples of fixed-point decimal constants:

Fixed-Point Constant	(size, precision)
1234567890123456D	(16, 0)
12345678.12345678D	(16, 8)
12345678.d	(8, 0)
.1234567890d	(10, 10)
12345.99d	(7, 2)
000123.990d	(9, 3)
0.00D	(3, 2)

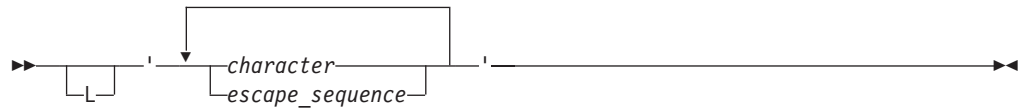
For more information on fixed-point decimal data types, see the *z/OS C/C++ Programming Guide*.

Literals

Character Literals

A *character literal* contains a sequence of characters or escape sequences enclosed in single quotation mark symbols, for example 'c'. A character literal may be prefixed with the letter L, for example L'c'. A character literal without the L prefix is an *ordinary character literal* or a *narrow character literal*. A character literal with the L prefix is a *wide character literal*. An ordinary character literal that contains more than one character or escape sequence (excluding single quotes (')), backslashes (\) or new-line characters) is a *multicharacter literal*.

Character literals have the following form:



At least one character or escape sequence must appear in the character literal. The characters can be from the source program character set, excluding the single quotation mark, backslash and new-line symbols. A character literal must appear on a single logical source line.

► **C++** A character literal that contains only one character has type **char**, which is an integral type. A multicharacter literal has type **int**.

► **C** A character literal has type **int**.

A wide character literal has type **wchar_t**. A multicharacter literal has type **int**.

The value of a narrow or wide character literal containing a single character is the numeric representation of the character in the character set used at run time. The value of a narrow or wide character literal containing more than one character or escape sequence is implementation-defined.

You can represent the double quotation mark symbol by itself, but you must use the backslash symbol followed by a single quotation mark symbol (\' escape sequence) to represent the single quotation mark symbol.

You can represent the new-line character by the \n new-line escape sequence.

You can represent the backslash character by the \\ backslash escape sequence.

The following are examples of character literals:

```
'a'
'\ '
L'0'
'('
```

The remainder of this section is discusses points of difference in the z/OS C/C++ implementation of character literals.

► **z/OS** The value of a character constant that contains a single character is the numeric representation of the character in the character set that is used at compile time. The value of a wide character constant containing a single multibyte character

is the code for that character, as defined by the `mbtowc()` function. If the character constant contains more than one character, the last 4 bytes represent the character constant. In z/OS C++, a character constant can contain only one character.

In z/OS C, a character constant has type `int`. In z/OS C++, a character constant has type `char`.

A wide character constant has type `wchar_t`, and is used to represent multibyte characters. Multibyte characters represent characters that use more than one byte for their encoding. Each multibyte character requires up to 4 bytes for its encoding.

RELATED REFERENCES

- “char and wchar_t Type Specifiers” on page 45

String Literals

A *string literal* contains a sequence of characters or escape sequences enclosed in double quotation mark symbols.



A string literal with the prefix **L** is a *wide string literal*. A string literal without the prefix **L** is an *ordinary* or *narrow string literal*.

The following are examples of string literals:

```

char
titles[ ] = "Handel's \"Water Music\"";
char *mail_addr = "Last Name   First Name   MI   Street Address \
    City   Province   Postal code ";
char *temp_string = "abc" "def" "ghi"; /* *temp_string = "abcdefghi\0" */
wchar_t *wide_string = L"longstring";

```

A null (`'\0'`) character is appended to each string. For a wide string literal, the value `'\0'` of type **wchar_t** is appended. By convention, programs recognize the end of a string by finding the null character.

Multiple spaces contained within a string literal are retained.

To continue a string on the next line, use the line continuation sequence (`\` symbol immediately followed by a new-line character). A carriage return must immediately follow the backslash. In the following example, the string literal second causes a compile-time error.

```

char *first = "This string continues onto the next\
line, where it ends."; /* compiles successfully. */
char *second = "The comment makes the \ /* continuation symbol */
invisible to the compiler."; /* compilation error. */

```

Another way to continue a string is to have two or more consecutive strings. Adjacent string literals will be concatenated to produce a single string. If a wide string literal and a narrow string literal are adjacent to each other, the resulting behavior is undefined. The following example demonstrates this:

Literals

```
"hello " "there"    /* is equivalent to "hello there"    */
"hello " L"there"    /* the behavior is undefined        */
"hello" "there"      /* is equivalent to "hellothere"      */
```

Characters in concatenated strings remain distinct. For example, the strings `"\xab"` and `"3"` are concatenated to form `"\xab3"`. However, the characters `\xab` and `3` remain distinct and are not merged to form the hexadecimal character `\xab3`.

Following any concatenation, `'\0'` of type **char** is appended at the end of each string. C++ programs find the end of a string by scanning for this value. For a wide string literal, `'\0'` of type **wchar_t** is appended. For example:

```
char *first = "Hello ";          /* stored as "Hello \0"          */
char *second = "there";          /* stored as "there\0"          */
char *third = "Hello " "there";  /* stored as "Hello there\0"    */
```

C++ The type of a narrow string literal is array of **const char** and the type of a wide string literal is array of **const wchar_t**. Both types have **static** storage duration.

C The type of a narrow string literal is array of **char** and the type of a wide string literal is array of **wchar_t**.

Use the escape sequence `\n` to represent a new-line character as part of the string. Use the escape sequence `\\` to represent a backslash character as part of the string. You can represent a single quotation mark symbol either by itself or with the escape sequence `'`. You must use the escape sequence `"` to represent a double quotation mark.

CCNX02K

```
/**
 ** This example illustrates escape sequences in string literals
 **/

#include <iostream>
using namespace std;

int main ()
{
    char *s ="Hi there! \n";
    cout << s;
    char *p = "The backslash character \\\.";
    cout << p << endl;
    char *q = "The double quotation mark \".\n";
    cout << q ;
}
```

This program produces the following output:

```
Hi there!
The backslash character \.
The double quotation mark ".
```

String Literals (z/OS)

z/OS This section describes considerations that are specific to z/OS C and C++ in regard to string literals.

In C, a character string literal has type *array of char* and static storage duration, whereas in C++, it has type *array of const char* and static storage duration. In C, a

wide character string literal has type *array of* `wchar_t` and static storage duration, whereas in C++, it has type *array of* `const wchar_t` and static storage duration.

You should be careful when modifying string literals because the resulting behavior depends on whether your strings are stored in read/write static memory. Both C strings and C++ strings are read-only by default.

Use the `#pragma strings` directive or the `R0STRING` compiler option to change the default storage for string literals. “strings” on page 256 describes the `#pragma strings` directive.

When a string literal appears more than once in the program source, how that string is stored depends on whether strings are *read-only* or *writable*. By default, the compiler considers strings to be *read-only*. z/OS C/C++ may allocate only one location for a *read-only* string; all occurrences will refer to that one location. However, that area of storage is potentially write-protected. If strings are *writable*, then each occurrence of the string will have a separate, distinct storage location that is always modifiable.

RELATED REFERENCES

- “char and wchar_t Type Specifiers” on page 45
- “volatile and const Qualifiers” on page 69
- “static Storage Class Specifier” on page 42

Boolean Literals

C++ There are only two boolean literals: **true** and **false**. These literals have type **bool** and are not lvalues.

RELATED REFERENCES

- “Boolean Variables” on page 46
- “Lvalues and Rvalues” on page 99

Chapter 3. Declarations

A *declaration* establishes the names and characteristics of data objects and functions used in a program. A *definition* allocates storage for data objects or specifies the body for a function. When you define a type, no storage is allocated.

Declarations Overview

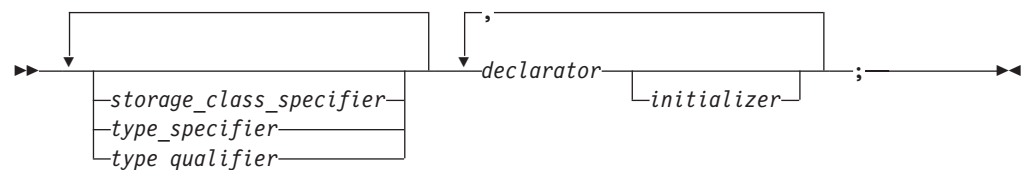
Declarations determine the following properties of data objects and their identifiers:

- Scope, which describes the visibility of an identifier in a block or source file.
- Linkage, which describes the association between two identical identifiers.
- Type, which describes the kind of data the object is to represent.

The lexical order of elements of a declaration for a data object is as follows:

- Storage duration and linkage specification
- Type specification
- Declarators, which introduce identifiers and make use of type qualifiers and storage qualifiers
- Initializers, which initialize storage with initial values

All data declarations have the form:



The following table shows examples of declarations and definitions. The identifiers declared in the first column do not allocate storage; they refer to a corresponding definition. In the case of a function, the corresponding definition is the code or body of the function. The identifiers declared in the second column allocate storage; they are both declarations and definitions.

Declarations	Declarations and Definitions
<code>extern double pi;</code>	<code>double pi = 3.14159265;</code>
<code>float square(float x);</code>	<code>float square(float x) { return x*x; }</code>
<code>struct payroll;</code>	<code>struct payroll { char *name; float salary; } employee;</code>

RELATED REFERENCES

- “Scope” on page 1
- “Program Linkage” on page 5
- “Storage Class Specifiers” on page 34
- “Type Specifiers” on page 44
- “Chapter 4. Declarators” on page 73
- “Initializers” on page 79
- “Chapter 7. Functions” on page 153

Objects

An *object* is a region of storage that contains a value or group of values. Each value can be accessed using its identifier or a more complex expression that refers to the object. In addition, each object has a unique *data type*. Both the identifier and data type of an object are established in the object *declaration*.

The data type of an object determines the initial storage allocation for that object and the interpretation of the values during subsequent access. It is also used in any type-checking operations.

Both C and C++ have built-in, or *fundamental* data types and user-defined data types. Standard data types include signed and unsigned integers, floating-point numbers, and characters. User-defined types include enumerations, structures, unions, and classes.

An instance of a class type is commonly called a *class object*. The individual class members are also called objects. The set of all member objects comprises a class object.

RELATED REFERENCES

- “Chapter 12. Classes” on page 283

Storage Class Specifiers

The storage class specifier used within the declaration determines whether:

- The object has internal, external, or no linkage
- The object is to be stored in memory or in a register, if available
- The object receives the default initial value 0 or an indeterminate default initial value
- The object can be referenced throughout a program or only within the function, block, or source file where the variable is defined
- The storage duration for the object is *static* (storage is maintained throughout program run time) or *automatic* (storage is maintained only during the execution of the block where the object is defined)


For a function, the storage class specifier determines the linkage of the function.

Declarations with the **auto** or **register** storage- class specifier result in automatic storage. Those with the **static** storage-class specifier result in static storage.

Most local declarations that do not include the **extern** storage-class specifier allocate storage; however, function declarations and type declarations do not allocate storage.

The only storage-class specifiers allowed in a namespace or global scope declaration are **static** and **extern**.

This section describes the following storage class specifiers:

- **auto**
- **extern**
-  **mutable**
- **register**
- **static**
- **typedef**

RELATED REFERENCES

- “auto Storage Class Specifier”
- “extern Storage Class Specifier” on page 37
- “mutable Storage Class Specifier” on page 40
- “register Storage Class Specifier” on page 41
- “static Storage Class Specifier” on page 42
- “typedef” on page 43
- “Program Linkage” on page 5

auto Storage Class Specifier

The **auto** storage class specifier lets you declare a variable with *automatic storage*. A variable *x* that has automatic storage is deleted when the block in which *x* was declared exits.

You can only apply the **auto** storage class specifier to names of variables declared in a block or to names of function parameters. However, these names by default have automatic storage. Therefore the storage class specifier **auto** is usually redundant in a data declaration.

Initialization

You can initialize any **auto** variable except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C or C++ expression. For structure and union members, the initial value must be a valid constant expression if an initializer list is used. The object is then set to that initial value each time the program block that contains the object’s definition is entered.

Note that if you use the **goto** statement to jump into the middle of a block, automatic variables within that block are not initialized.

Storage

Objects with the **auto** storage class specifier have automatic storage duration. Each time a block is entered, storage for **auto** objects defined in that block is made available. When the block is exited, the objects are no longer available for use.

If an **auto** object is defined within a function that is recursively invoked, memory is allocated for the object at each invocation of the block.

Examples of auto Storage Class

The following program shows the scope and initialization of auto variables. The function `main` defines two variables, each named `auto_var`. The first definition occurs on line 10. The second definition occurs in a nested block on line 13. While the nested block is running, only the `auto_var` that is created by the second definition is available. During the rest of the program, only the `auto_var` that is created by the first definition is available.

CCNRAAF

```

1  /*****
2  ** Example illustrating the use of auto variables **
3  *****/
4
5  #include <stdio.h>
6
7  int main(void)
```

Storage Class Specifiers

```
8  {
9      void call_func(int passed_var);
10     auto int auto_var = 1; /* first definition of auto_var */
11
12     {
13         int auto_var = 2;    /* second definition of auto_var */
14         printf("inner auto_var = %d\n", auto_var);
15     }
16     call_func(auto_var);
17     printf("outer auto_var = %d\n", auto_var);
18     return 0;
19 }
20
21 void call_func(int passed_var)
22 {
23     printf("passed_var = %d\n", passed_var);
24     passed_var = 3;
25     printf("passed_var = %d\n", passed_var);
26 }
```

This program produces the following output:

```
inner auto_var = 2
passed_var = 1
passed_var = 3
outer auto_var = 1
```

The following example uses an array that has the storage class `auto` to pass a character string to the function `sort`. The function `sort` receives the address of the character string, rather than the contents of the array. The address enables `sort` to change the values of the elements in the array.

CCNRAAG

```
/******
** Sorted string program -- this example passes an array name **
** to a function                                           **
*****/

#include <stdio.h>
#include <string.h>

int main(void)
{
    void sort(char *array, int n);
    char string[75];
    int length;

    printf("Enter letters:\n");
    scanf("%74s", string);
    length = strlen(string);
    sort(string, length);
    printf("The sorted string is: %s\n", string);

    return(0);
}

void sort(char *array, int n)
{
    int gap, i, j, temp;

    for (gap = n / 2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j = i - gap; j >= 0 && array[j] > array[j + gap];
                j -= gap)
            {
                temp = array[j];
```

```

        array[j] = array[j + gap];
        array[j + gap] = temp;
    }
}

```

When you run the program, interaction with the program could produce:

Output Enter letters:

Input zyfab

Output The sorted string is: abfyz

RELATED REFERENCES

- “Block Statement” on page 179
- “goto Statement” on page 193
- “Function Declarations” on page 154

extern Storage Class Specifier

The **extern** storage class specifier lets you declare objects and functions that several source files can use. All object declarations that occur outside a function and that do not contain a storage class specifier declare identifiers with external linkage. All function definitions that do not specify a storage class define functions with external linkage.

> C++ In C++, the **extern** storage class specifier can only be applied to names of objects or functions. Using the **extern** specifier with type declarations is illegal.

An **extern** variable, function definition, or declaration also makes the described variable or function usable by the succeeding part of the current source file. This declaration does not replace the definition. The declaration is used to describe the variable that is externally defined.

If a declaration for an identifier already exists at file scope, any **extern** declaration of the same identifier found within a block refers to that same object. If no other declaration for the identifier exists at file scope, the identifier has external linkage.

An **extern** declaration can appear outside a function or at the beginning of a block. If the declaration describes a function or appears outside a function and describes an object with external linkage, the keyword **extern** is optional.

If you do not specify a storage class specifier, the function has external linkage. It is an error to include a declaration for the same function with the storage class specifier **static** before the declaration with no storage class specifier because of the incompatible declarations. Including the **extern** storage class specifier on the original declaration is valid and the function has internal linkage.

> C++ In C++, an **extern** declaration cannot appear in class scope.

Initialization

You can initialize any object with the **extern** storage class specifier at namespace (or global scope in C). You can initialize an **extern** object with an initializer that must either:

- Appear as part of the definition and the initial value must be described by a constant expression. OR

Storage Class Specifiers

- Reduce to the address of a previously declared object with static storage duration. You may modify this object with pointer arithmetic. (In other words, you may modify the object by adding or subtracting an integral constant expression.)

If you do not explicitly initialize an **extern** variable, its initial value is zero of the appropriate type. Initialization of an **extern** object is completed by the time the program starts running.

Storage

All **extern** objects have static storage duration. Memory is allocated for **extern** objects before the **main** function begins running. When the program finishes running, the storage is freed.

RELATED REFERENCES

- “External Linkage” on page 6
- “Internal Linkage” on page 5
- “static Storage Class Specifier” on page 42
- “Class Scope” on page 3
- “Chapter 10. Namespaces” on page 261

extern Storage Class Specifier (z/OS)

C++ In z/OS C++, you can declare functions with the following:

Linkage	By specifying
C	extern "C"
C++	extern "C++"
OS	extern "OS"
PLI	extern "PLI"
builtin	extern "builtin"
COBOL	extern "COBOL"
FORTRAN	extern "FORTRAN"
OS_DOWNSTACK	extern "OS_DOWNSTACK"
OS_UPSTACK	extern "OS_UPSTACK"
OS_NOSTACK	extern "OS_NOSTACK"
OS31_NOSTACK	extern "OS31_NOSTACK"
REFERENCE	extern "REFERENCE"

There are some limitations to using extern to specify non-C++ linkage for a function. While the C++ language supports overloading, other languages do not. The implications of this are:

- You cannot overload a function that has non-C++ linkage:

```
extern "FORTRAN">{int func(int);}
extern "FORTRAN">{int func(int,int);} // not allowed-compiler
// will issue an error message
```


- You cannot declare a function with a linkage specification if you have already used the same function name in a declaration without a linkage specification:

```
int func(int);
extern "FORTRAN">{int func(int,int);} // not allowed-compiler
// will issue an error message
```

- You can overload a function as long as it has C++ (default) linkage. Therefore, z/OS C/C++ allows the following series of statements:

```
extern "FORTRAN">{int func(int,int);}
int func(int); // function with C++ linkage
int func(int,int); // overloaded function with C++ linkage
```

- You cannot redefine a function that has a linkage specification:

```
extern func(int);
extern "FORTRAN">{int func(int,int);} // not allowed-compiler
// will issue an error message
```

The following fragments illustrate the use of extern "C" :

```
extern "C" int cf(); //declare function cf to have C linkage

extern "C" int (*c_fp)(); //declare a pointer to a function,
// called c_fp, which has C linkage

extern "C" {
    typedef void(*cfp_T)(); //create a type pointer to function with C
    // linkage
    void cfn(); //create a function with C linkage
    void (*cfp)(); //create a pointer to a function, with C
    // linkage
}
```

Linkage compatibility affects all C library functions that accept a user function pointer as a parameter. Use the extern "C" linkage specification to ensure that the declared linkages are the same. An example of these library functions is `qsort()`. See other z/OS documentation for more information.

The following example fragment uses extern "C" with `qsort()`.

```
#include <stdlib.h>

// function to compare table elements
extern "C" int TableCmp(const void *, const void *); // C linkage
extern void * GenTable(); // C++ linkage

int main() {
    void *table;

    table = GenTable(); // generate table
    qsort(table, 100, 15, TableCmp); // sort table, using TableCmp
    // and C library routine qsort();
}
```

C++ In z/OS C++, an extern declaration cannot appear in class scope.

Controlling External Static (z/OS)

z/OS Certain program variables with the extern storage class may be constant and never be updated. If this is the case, it is not necessary to have a copy of these variables made for every user of the program. In addition, there may be a need to share constant program variables between C and another language.

Storage Class Specifiers

Examples of extern Storage Class (z/OS)

z/OS The following program fragment shows how to force an external program variable to be part of a program that includes executable code and constant data. It uses the `#pragma variable(varname, NORENT)` directive:

```
#pragma variable(rates, NORENT)
extern float rates[5] = { 3.2, 83.3, 13.4, 3.6, 5.0
};

extern float totals[5];

int main(void) {
    :
}
}
```

In this example, you compile the source file with the `RENT` option. The executable code includes the variable `rates` as you specify the `#pragma variable(rates, NORENT)`. The writable static includes the variable `totals`. Each user has a personal copy of the array `totals`, and all users of the program share the array `rates`. This sharing may yield a performance and storage benefit.

The `#pragma variable(varname, NORENT)` does not apply to, and has no effect on, program variables with the `static` storage class. `z/OS C/C++` always includes program variables with the `static` storage class with the `writable static`. An informational message appears if you write to a nonreentrant variable when you specify the `C CHECKOUT` compile-time option.

When you specify `#pragma variable(varname, NORENT)` for a variable, ensure that your program never writes to this variable. Program exceptions or unpredictable program behavior may result should this be the case. In addition, you must include `#pragma variable(varname, NORENT)` in every source file where you reference or define the variable.

Refer to *z/OS C/C++ User's Guide* for more information on the `RENT` and `NORENT` compile-time options.

mutable Storage Class Specifier

C++ The *mutable* storage class specifier is used only on a class data member to make it modifiable even though the member is part of an object declared as `const`. You cannot use the `mutable` specifier with names declared as **static** or **const**, or reference members.

```
class A
{
    public:
        A() : x(4), y(5) { };
        mutable int x;
        int y;
};

int main()
{
    const A var2;
    var2.x = 345;
    // var2.y = 2345;
}
```

In this example, the compiler would not allow the assignment `var2.y = 2345` because `var2` has been declared as `const`. The compiler will allow the assignment `var2.x = 345` because `A::x` has been declared as **mutable**.

RELATED REFERENCES

- “static Storage Class Specifier” on page 42
- “volatile and const Qualifiers” on page 69
- “References” on page 92

register Storage Class Specifier

The **register** storage class specifier indicates to the compiler that a heavily used variable (such as a loop control variable) within a local scope data definition or a parameter declaration should be allocated a register to minimize access time.

It is equivalent to the **auto** storage class except that the compiler places the object, if possible, into a machine register for faster access. An object having the **register** storage class specifier must be defined within a block or declared as a parameter to a function.

Initialization

You can initialize any **register** object except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C or C++ expression. For structure and union members, the initial value must be a valid constant expression if an initializer list is used. The object is then set to that initial value each time the program block that contains the object’s definition is entered.

Storage



Objects with the **register** storage class specifier have automatic storage duration. Each time a block is entered, storage for **register** objects defined in that block are made available. When the block is exited, the objects are no longer available for use.

If a **register** object is defined within a function that is recursively invoked, the memory is allocated for the variable at each invocation of the block.

The **register** storage class specifier indicates that the object is heavily used and indicates to the compiler that the value of the object should reside in a machine register. Because of the limited size and number of registers available on most systems, few variables can actually be put in registers.

If the compiler does not allocate a machine register for a **register** object, the object is treated as having the storage class specifier **auto**.

Restrictions

  You cannot use the **register** storage class specifier in namespace scope (or global scope in C) data declarations. In C programs, you cannot apply the address (&) operator to **register** variables. However, C++ lets you take the address of an object with the **register** storage class. For example:

```
register int i;  
int* b = &i;    // valid in C++, but not in C
```

Storage Class Specifiers

RELATED REFERENCES

- “Local Scope” on page 1
- “auto Storage Class Specifier” on page 35
- “References” on page 92

static Storage Class Specifier

The **static** storage class specifier lets you define objects with static storage duration and internal linkage, or to define functions with internal linkage.

The **static** storage class specifier can only be applied to the following names:

- Objects
- Functions
- Class members
- Anonymous unions

You cannot declare any of the following as **static**:

- Type declarations
- Function declarations within a block
- Function parameters

Objects with the **static** storage class specifier have *static storage duration*. The storage for a **static** variable is made available when the program begins running. When the program finishes running, the memory is freed.

For example, suppose a static variable *x* has been declared in function *f()*. When the program exits the scope of *f()*, *x* is not destroyed. The following example demonstrates this:

```
#include <stdio.h>

int f(void) {
    static int i = 0;
    i++;
    return i;
}

int main(void) {
    int j;
    for (j = 0; j < 5; j++) {
        printf("Value of f(): %d\n", f());
    }
    return 0;
}
```

The following is the output of the above example:

```
Value of f(): 1
Value of f(): 2
Value of f(): 3
Value of f(): 4
Value of f(): 5
```

Because *i* is a static variable, it is not reinitialized to 0 on successive calls to *f()*.

If you explicitly declare an object, reference, function, or function template **static** in namespace or global scope, that entity will have internal linkage; you cannot use that entity in other translation units.

Initialization

► **C** You initialize a **static** object with a constant expression, or an expression that reduces to the address of a previously declared **extern** or **static** object, possibly modified by a constant expression.

► **C++** You may initialize a **static** object with a non-constant expression.

► **C** If you do not explicitly initialize a static (or external) variable, it will have a value of zero of the appropriate type.

► **C++** A static object of class type will use the default constructor if you do not initialize it.

Automatic and register variables that are not initialized will have undefined values.

RELATED REFERENCES

- “Internal Linkage” on page 5
- “extern Storage Class Specifier” on page 37
- “Objects” on page 34
- “Class Member Lists” on page 293
- “Anonymous Unions” on page 62

typedef

A **typedef** declaration lets you define your own identifiers that can be used in place of type specifiers such as **int**, **float**, and **double**. A **typedef** declaration does not reserve storage. The names you define using **typedef** are not new data types. They are synonyms for the data types or combinations of data types they represent.

► **C++** In C++, a **typedef** name must be different from any class type name declared within the same scope. If the **typedef** name is the same as a class type name, it can only be so if that **typedef** is a synonym of the class name. This condition is not the same as in C.

When an object is defined using a **typedef** identifier, the properties of the defined object are exactly the same as if the object were defined by explicitly listing the data type associated with the identifier.

Examples of typedef Declarations

The following statements declare **LENGTH** as a synonym for **int** and then use this **typedef** to declare **length**, **width**, and **height** as integer variables:

```
typedef int LENGTH;
LENGTH length, width, height;
```

The following declarations are equivalent to the above declaration:

```
int length, width, height;
```

Similarly, **typedef** can be used to define a class type (structure, union, or C++ class). For example:

```
typedef struct {
    int scruples;
    int drams;
    int grains;
} WEIGHT;
```

Storage Class Specifiers

The structure `WEIGHT` can then be used in the following declarations:

```
WEIGHT chicken, cow, horse, whale;
```

C++ A C++ class defined in a **typedef** without being named is given a dummy name and the **typedef** name for linkage. Such a class cannot have constructors or destructors. For example:

```
typedef class {  
    Trees();  
} Trees;
```

Here the function `Trees()` is an ordinary member function of a class whose type name is unspecified. In the above example, `Trees` is an alias for the unnamed class, not the class type name itself, so `Trees()` cannot be a constructor for that class.

RELATED REFERENCES

- “Type Specifiers”
- “Arrays” on page 86
- “Structures” on page 51
- “Unions” on page 59
- “Chapter 12. Classes” on page 283
- “Constructors and Destructors Overview” on page 341

Type Specifiers

C++ In C++, types must be declared in declarations. They may not be declared in expressions.

Type specifiers indicate the type of the object or function being declared. The following are the available kinds of type specifiers:

- Simple type specifiers
- **C++** Class specifiers
- Enumerated specifiers
- **C++** Elaborated type specifiers
- **const** and **volatile** qualifiers

RELATED REFERENCES

- “Simple Type Specifiers”
- “Declaring Class Types” on page 283
- “Enumerations” on page 65
- “Type Specifiers”
- “Type Specifiers”

Simple Type Specifiers

A *simple type specifier* either specifies a (previously declared) user-defined type or a *fundamental type*. A fundamental type is a type built-in to the language. The following describes how the fundamental types are categorized:

- Arithmetic types
 - Integral types
 - **C++** **bool**
 - **char**
 - **wchar_t**

- Signed integer types
 - **signed char**
 - **short int**
 - **int**
 - **long int**
- Unsigned integer types
 - **unsigned char**
 - **unsigned short int**
 - **unsigned int**
 - **unsigned long int**
- Floating-point types
 - **float**
 - **double**
 - **long double**
- **void**

RELATED REFERENCES

- “char and wchar_t Type Specifiers”
- “Boolean Variables” on page 46
- “Floating-Point Variables” on page 47
- “Integer Variables” on page 49
- “void Type” on page 50

char and wchar_t Type Specifiers

The **char** specifier has the following syntax:



The **char** specifier is an integral type.

A **char** has enough storage to represent a character from the basic character set. The amount of storage allocated for a **char** is implementation-dependent.

You initialize a variable of type **char** with a character literal (consisting of one character) or with an expression that evaluates to an integer.

Use **signed char** or **unsigned char** to declare numeric variables that occupy a single byte.

C++ For the purposes of distinguishing overloaded functions, a C++ **char** is a distinct type from **signed char** and **unsigned char**.

z/OS There are three character data types: char, signed char, and unsigned char in the z/OS C and C++ implementation. These three data types are not compatible. If you specify `LANGlvl(ANSI)`, the C compiler recognizes char, unsigned char, and signed char as distinct types. They are always distinct types in C++.

z/OS The character data types provide enough storage to hold any member of the character set your program uses at run time. The amount of storage that is

Type Specifiers

allocated for a char is implementation-dependent. The z/OS C/C++ compiler represents a character by 8 bits, as defined in the CHAR_BIT macro in the <limits.h> header.

➤ **z/OS** The default character type behaves like an unsigned char. To change this default, use #pragma chars, described in “chars” on page 224.

➤ **z/OS** In the z/OS C and C++ implementation, it does not matter whether a char data object is signed or unsigned, you can declare the object as having the data type char. Otherwise, explicitly declare signed char or unsigned char. When a char (signed or unsigned) is widened to an int, its value is preserved.

The wchar_t Type Specifier

The **wchar_t** type specifier has enough storage to represent a wide character literal. (A character literal that is prefixed with the letter L, for example L'x', is a wide character literal).

The **wchar_t** type specifier is an integral type.

Examples of the char Type Specifier

The following example defines the identifier end_of_string as a constant object of type **char** having the initial value \0 (the null character):

```
const char end_of_string = '\0';
```

The following example defines the **unsigned char** variable switches as having the initial value 3:

```
unsigned char switches = 3;
```

The following example defines string_pointer as a pointer to a character:

```
char *string_pointer;
```

The following example defines name as a pointer to a character. After initialization, name points to the first letter in the character string "Johnny":

```
char *name = "Johnny";
```

The following example defines a one-dimensional array of pointers to characters. The array has three elements. Initially they are a pointer to the string "Venus", a pointer to "Jupiter", and a pointer to "Saturn":

```
static char *planets[ ] = { "Venus", "Jupiter", "Saturn" };
```

RELATED REFERENCES

- “Character Literals” on page 28
- “Integer Variables” on page 49

Boolean Variables

➤ **C++** Use the type specifier bool and the literals true and false to make boolean logic tests. A *boolean logic test* is used to express the results of a logical operation. For example:

```
bool f(int a, int b)
{
    return a==b;
}
```


If *a* and *b* have the same value, *f()* returns true. If not, *f()* returns false.

Variables of type **bool** can hold either one of two values: true or false. An rvalue of type **bool** can be promoted to an integral type. A **bool** rvalue of false is promoted to the value 0 and a **bool** rvalue of true is promoted to the value 1.

RELATED REFERENCES

- “Boolean Literals” on page 31
- “Integer Variables” on page 49

Floating-Point Variables

There are three types of floating-point variables:

- **float**
- **double**
- **long double**

To declare a data object that is a floating-point type, use the following **float** specifier:



The declarator for a simple floating-point declaration is an identifier. Initialize a simple floating-point variable with a float constant or with a variable or expression that evaluates to an integer or floating-point number. The storage class of a variable determines how you initialize the variable.

z/OS Note that z/OS C/C++ supports IEEE binary floating-point variables as well as IBM S/390 hexadecimal floating-point variables. For details on the **FLOAT** option, please see *z/OS C/C++ User's Guide*.

Examples of Floating-Point Data Types

The following example defines the identifier *pi* as an object of type **double**:

```
double pi;
```

The following example defines the **float** variable *real_number* with the initial value 100.55:

```
static float real_number = 100.55f;
```

Note: If you do not add the **f** suffix to a floating-point literal, that number will be of type **double**. If you initialize an object of type **float** with an object of type **double**, the compiler will implicitly convert the object of type **double** to an object of type **float**.

The following example defines the **float** variable *float_var* with the initial value 0.0143:

```
float float_var = 1.43e-2f;
```

The following example declares the **long double** variable *maximum*:

```
extern long double maximum;
```

Type Specifiers

The following example defines the array `table` with 20 elements of type **double**:

```
double table[20];
```

RELATED REFERENCES

- “Floating-Point Literals” on page 25
- “Assignment Expressions” on page 134

Fixed-Point Decimal Data Types (z/OS C Only)

z/OS Use the type specifier *decimal(n,p)* to declare fixed-point decimal variables and to initialize them with fixed-point decimal constants. For this type specifier, *decimal* is a macro that is defined in `<decimal.h>`. Remember to include `<decimal.h>` if you use fixed-point decimals in your program.

Fixed-point decimal types are classified as arithmetic types. The *decimal(n,p)* type specifier designates a decimal number with *n* digits, and *p* decimal places. *n* is the total number of digits for the integral and decimal parts combined. *p* is the number of digits for the decimal part only. For example, *decimal(5,2)* represents a number, such as, 123.45 where *n*=5 and *p*=2. The value for *p* is optional. If you leave it out, the default value is 0.

In the type specifier, *n* and *p* have a range of allowed values according to the following rules:

```
p <= n
1 <= n <= DEC_DIG
0 <= p <= DEC_PRECISION
```

Note: `<decimal.h>` defines `DEC_DIG` (the maximum number of digits *n*) and `DEC_PRECISION` (the maximum precision *p*). Currently, it uses a maximum of 31 digits for both limits.

The following examples show how to declare a variable as a fixed-point decimal data type:

```
decimal(10,2) x;
decimal(5,0) y;
decimal(5) z;
decimal(18,10) *ptr;
decimal(8,2) arr[100];
```

In the previous example:

- *x* can have values between -99999999.99D and +99999999.99D.
- *y* and *z* can have values between -99999D and +99999D.
- *ptr* is a pointer to type *decimal(18,10)*.
- *arr* is an array of 100 elements, where each element is of type *decimal(8,2)*.


The fixed-point decimal type specifier has the form:

►► decimal (— *constant_expression* — [, — *constant_expression* —]) —►►

z/OS C/C++ evaluates the first *constant_expression* as a positive integral constant expression. The second *constant_expression* is optional. If you leave it out, the default value is 0. The type specifiers, *decimal(n,0)* and *decimal(n)* are type-compatible.

Integer Variables

Integer variables fall into the following categories:


- integral types
 -  **bool**
 - **char**
 - **wchar_t**
 - signed integer types
 - **signed char**
 - **short int**
 - **int**
 - **long int**
 - unsigned integer types
 - **unsigned char**
 - **unsigned short int**
 - **unsigned int**
 - **unsigned long int**

The default integer type for a bit field is **unsigned**.

The amount of storage allocated for integer data is implementation-dependent.

The **unsigned** prefix indicates that the object is a nonnegative integer. Each unsigned type provides the same size storage as its signed equivalent. For example, **int** reserves the same storage as **unsigned int**. Because a signed type reserves a sign bit, an unsigned type can hold a larger positive integer than the equivalent signed type.

The declarator for a simple integer definition or declaration is an identifier. You can initialize a simple integer definition with an integer constant or with an expression that evaluates to a value that can be assigned to an integer. The storage class of a variable determines how you can initialize the variable.

 When the arguments in overloaded functions and overloaded operators are integer types, two integer types that both come from the same group are not treated as distinct types. For example, you cannot overload an **int** argument against a **signed int** argument.

Examples of Integer Data Types

The following example defines the **short int** variable `flag`:

```
short int flag;
```

The following example defines the **int** variable `result`:

```
int result;
```

The following example defines the **unsigned long int** variable `ss_number` as having the initial value 438888834 :

```
unsigned long ss_number = 438888834ul;
```

RELATED REFERENCES

- “Integer Literals” on page 24
- “Decimal Integer Literals” on page 24
- “Octal Decimal Literals” on page 25

Type Specifiers

- “Hexadecimal Decimal Literals” on page 25
- “Chapter 11. Overloading” on page 269
- “z/OS Integer Variables”

z/OS Integer Variables

z/OS In the z/OS C and C++ implementation, integer variables fall into the following categories:

- short int or short or signed short int or signed short
- signed int or int
- long int or long or signed long int or signed long
- long long int or long long or signed long long int or signed long long
- unsigned short int or unsigned short
- unsigned or unsigned int
- unsigned long int or unsigned long
- unsigned long long int or unsigned long long

C++ z/OS C/C++ supports the long long data type for language levels other than ANSI by default.

You can also control the support for long long using the `LONGLONG` suboption of `LANGLVL`. For example, specifying `LANGLVL(ANSI, LONGLONG)` would add the long long data type to the ISO language level. Please refer to *z/OS C/C++ User's Guide* for information on using the `LANGLVL` option.

The default integer type for a bit field is unsigned. The amount of storage that is allocated for integer data is implementation- dependent.

z/OS C/C++ provides three sizes of integer data types. Objects that are of type `short` have a length of 2 bytes of storage. Objects that are of type `long` have a length of 4 bytes of storage. Objects that are of type `long long` have a length of 8 bytes of storage. An `int` data type represents the most efficient data storage size on the system (the word-size of the machine) and receives 4 bytes of storage.

void Type

The **void** data type always represents an empty set of values. The only object that can be declared with the type specifier **void** is a pointer.

When a function does not return a value, you should use **void** as the type specifier in the function definition and declaration. An argument list for a function taking no arguments is **void**.

You cannot declare a variable of type **void**, but you can explicitly convert any expression to type **void**. The resulting expression can only be used as one of the following:

- An expression statement
- The left operand of a comma expression
- The second or third operand in a conditional expression.

Example of void Type

In the following example, the function `find_max` is declared as having type **void**.

Note: **C** The use of the `sizeof` operator in the line `find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));` is a standard method of determining the number of elements in an array.

CCNRAAM

```

/**
** Example of void type
**/
#include <stdio.h>

/* declaration of function find_max */
extern void find_max(int x[ ], int j);

int main(void)
{
    static int numbers[ ] = { 99, 54, -102, 89};

    find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));

    return(0);
}

void find_max(int x[ ], int j)
{ /* begin definition of function find_max */
    int i, temp = x[0];

    for (i = 1; i < j; i++)
    {
        if (x[i] > temp)
            temp = x[i];
    }
    printf("max number = %d\n", temp);
} /* end definition of function find_max */

```

RELATED REFERENCES

- “Pointers” on page 81
- “Comma Expression ,” on page 140
- “Conditional Expressions” on page 137
- “Function Declarations” on page 154

Structures

► C A *structure* contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have varied data types. Each data object in a structure is a *member* or *field*.

Use structures to group logically related objects. For example, to allocate storage for the components of one address, define the following variables:

```

int street_no;
char *street_name;
char *city;
char *prov;
char *postal_code;

```

To allocate storage for more than one address, group the components of each address by defining a structure data type and as many variables as you need to have the structure data type.

In the following example, line `int street_no;` through to `char *postal_code;` declare the structure tag address:

```

struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;

```

Type Specifiers

```
        char *postal_code;
    };
    struct address perm_address;
    struct address temp_address;
    struct address *p_perm_address = &perm_address;
```

The variables `perm_address` and `temp_address` are instances of the structure data type `address`. Both contain the members described in the declaration of `address`. The pointer `p_perm_address` points to a structure of `address` and is initialized to point to `perm_address`.

Refer to a member of a structure by specifying the structure variable name with the dot operator (`.`) or a pointer with the arrow operator (`->`) and the member name. For example, both of the following:

```
perm_address.prov = "Ontario";
p_perm_address -> prov = "Ontario";
```

assign a pointer to the string "Ontario" to the pointer `prov` that is in the structure `perm_address`.

All references to structures must be fully qualified. In the example, you cannot reference the fourth field by `prov` alone. You must reference this field by `perm_address.prov`.

Structures with identical members but different names are not compatible and cannot be assigned to each other.

Structures are not intended to conserve storage. If you need direct control of byte mapping, use pointers.

You cannot declare a structure with members of incomplete types.

C++ In C++ a structure is the same as a class except that its members and inheritance are public by default.

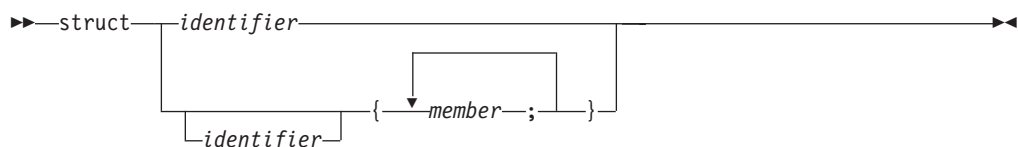
RELATED REFERENCES

- "Chapter 12. Classes" on page 283
- "Dot Operator `.`" on page 107
- "Arrow Operator `->`" on page 107
- "Incomplete Types" on page 71

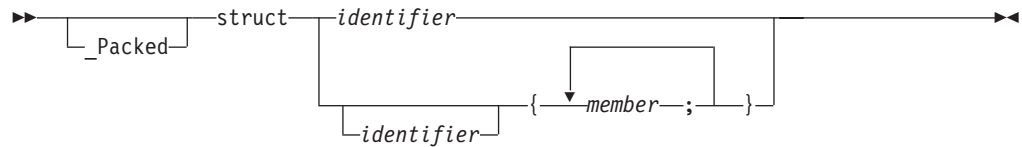
Declaring and Defining a Structure

C A *structure type definition* describes the members that are part of the structure. It contains the **struct** keyword followed by an optional identifier (the structure tag) and a brace-enclosed list of members.

A structure definition has the form:



z/OS A structure definition for the z/OS implementation has the following variation of that form:



A *structure declaration* has the same form as a structure definition except the declaration does not have a brace-enclosed list of members.

The keyword **struct** followed by the identifier (tag) names the data type. If you do not provide a tag name to the data type, you must put all variable definitions that refer to it within the declaration of the data type.

The list of members provides the data type with a description of the values that can be stored in the structure.

A structure data member definition has the form of a variable declaration. However you may declare a *bit-field* as a member for a structure. A member that does not represent a bit field can be of any data type and can have the **volatile** or **const** qualifier.

Identifiers used as structure or member names can be redefined to represent different objects in the same scope without conflicting. You cannot use the name of a member more than once in a structure type, but you can use the same member name in another structure type that is defined within the same scope.

You cannot declare a structure type that contains itself as a member, but you can declare a structure type that contains a pointer to itself as a member.

RELATED REFERENCES

- “Declaring and Using Bit Fields in Structures” on page 55
- “volatile and const Qualifiers” on page 69

Defining a Structure Variable

► **C** A structure variable definition contains an optional storage class keyword, the **struct** keyword, a structure tag, a declarator, and an optional identifier. The structure tag indicates the data type of the structure variable.

► C++ The keyword **struct** is optional in C++.

You can declare structures having any storage class. Most compilers, however, treat structures declared with the **register** storage class specifier as automatic structures.

RELATED REFERENCES

- “auto Storage Class Specifier” on page 35
- “register Storage Class Specifier” on page 41

Initializing Structures

C The initializer contains an = (equal sign) followed by a brace-enclosed comma-separated list of values. You do not have to initialize all members of a structure.

The following definition shows a completely initialized structure:

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};
static struct address perm_address =
    { 3, "Savona Dr.", "Dundas", "Ontario", "L4B 2A1"};
```

The values of perm_address are:

Member	Value
perm_address.street_no	3
perm_address.street_name	address of string "Savona Dr."
perm_address.city	address of string "Dundas"
perm_address.prov	address of string "Ontario"
perm_address.postal_code	address of string "L4B 2A1"

The following definition shows a partially initialized structure:

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};
struct address temp_address =
    { 44, "Knyvet Ave.", "Hamilton", "Ontario" };
```

The values of temp_address are:

Member	Value
temp_address.street_no	44
temp_address.street_name	address of string "Knyvet Ave."
temp_address.city	address of string "Hamilton"
temp_address.prov	address of string "Ontario"
temp_address.postal_code	value depends on the storage class.

Note: The initial value of uninitialized structure members like temp_address.postal_code depends on the storage class associated with the member.

Declaring Structure Types and Variables

C To define a structure type and a structure variable in one statement, put a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the statement.

For example:

```
static struct {
    int street_no;
    char *street_name;
```



```

char *city;
char *prov;
char *postal_code;
} perm_address, temp_address;

```

Because this example does not name the structure data type, `perm_address` and `temp_address` are the only structure variables that will have this data type. Putting an identifier after **struct**, lets you make additional variable definitions of this data type later in the program.

The structure type (or tag) cannot have the **volatile** qualifier, but a member or a structure variable can be defined as having the **volatile** qualifier.

For example:

```

static struct class1 {
    char descript[20];
    volatile long code;
    short complete;
} volatile file1, file2;
struct class1 subfile;

```

This example qualifies the structures `file1` and `file2`, and the structure member `subfile.code` as **volatile**.

RELATED REFERENCES

- “Initializing Structures” on page 54
- “Storage Class Specifiers” on page 34
- “volatile and const Qualifiers” on page 69

Declaring and Using Bit Fields in Structures

C **C++** A structure or a C++ class can contain *bit fields* that allow you to access individual bits. You can use bit fields for data that requires just a few bits of storage. A bit field declaration contains a type specifier followed by an optional declarator, a colon, a constant expression, and a semicolon. The constant expression specifies how many bits the field reserves.

Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized. A zero-width bit field causes the next field to be aligned on the next container boundary where the container is the same size as the underlying type as the bit field. A `_Packed` structure, which is a bit field of length 0, causes the next field to align on the next byte boundary.

The padding to the next container boundary only takes place if the zero-width bit field has the same underlying type as the preceding bit-field member. If the types are different, the zero-width bit field has no effect.

The maximum bit-field length is implementation dependent.

For portability, do not use bit fields greater than 32 bits in size.

The following restrictions apply to bit fields. You cannot:

- Define an array of bit fields
- Take the address of a bit field
- Have a pointer to a bit field
- Have a reference to a bit field

Type Specifiers

► C In C, you can declare a bit field as type **int**, **signed int**, or **unsigned int**. Bit fields of the type **int** are equivalent to those of type **unsigned int**.

For all implementations, the default integer type for a bit field is **unsigned**.

A bit field cannot have the **const** or **volatile** qualifier.

The following structure has three bit-field members `kingdom`, `phylum`, and `genus`, occupying 12, 6, and 2 bits respectively:

```
struct taxonomy {
    int kingdom : 12;
    int phylum : 6;
    int genus : 2;
};
```

► C++ Unlike C, C++ bit fields can be any integral type or enumeration type. When you assign a value that is out of range to a bit field, the low-order bit pattern is preserved and the appropriate bits are assigned.

If a series of bit fields does not add up to the size of an **int**, padding can take place. The amount of padding is determined by the alignment characteristics of the members of the structure.

► z/OS z/OS C/C++ determines the amount of padding by the alignment characteristics of the structure members. In some instances, bit fields can cross word boundaries.

The following example demonstrates padding, and is valid for all implementations. Suppose that an **int** occupies 4 bytes. The example declares the identifier `kitchen` to be of type `struct on_off`:

```
struct on_off {
    unsigned light : 1;
    unsigned toaster : 1;
    int count;          /* 4 bytes */
    unsigned ac : 4;
    unsigned : 4;
    unsigned clock : 1;
    unsigned : 0;
    unsigned flag : 1;
} kitchen ;
```

The structure `kitchen` contains eight members totalling 16 bytes. The following table describes the storage that each member occupies:

Member Name	Storage Occupied
<code>light</code>	1 bit
<code>toaster</code>	1 bit
(padding — 30 bits)	To the next int boundary
<code>count</code>	The size of an int (4 bytes)
<code>ac</code>	4 bits
(unnamed field)	1 bit, ► z/OS 4 bits
<code>clock</code>	1 bit
(padding — 23 bits)	To the next int boundary (unnamed field)

Member Name	Storage Occupied
flag	1 bit
(padding — 31 bits)	To the next int boundary


All references to structure fields must be fully qualified. For instance, you cannot reference the second field by `toaster`. You must reference this field by `kitchen.toaster`.


The following expression sets the `light` field to 1:

```
kitchen.light = 1;
```

When you assign to a bit field a value that is out of its range, the bit pattern is preserved and the appropriate bits are assigned. The following expression sets the `toaster` field of the `kitchen` structure to 0 because only the least significant bit is assigned to the `toaster` field:

```
kitchen.toaster = 2;
```

Declaring a Packed Structure (z/OS C/C++):  To qualify a C structure as packed, use the `_Packed` qualifier on the structure declaration.

 C++ does not support the `_Packed` qualifier. To change the alignment of C++ structures, use the `#pragma pack` directive, which is supported by both C and C++. Please refer to “pack” on page 249 for information on this directive.

Packed and nonpacked structures cannot be assigned to each other, regardless of their type.

RELATED REFERENCES

- “Chapter 12. Classes” on page 283
- “Arrays” on page 86
- “Address &” on page 116
- “Pointers” on page 81
- “References” on page 92

Example Program Using Structures

The following program finds the sum of the integer numbers in a linked list:

CCNRAAS

```
/**
 ** Example program illustrating structures using linked lists
 **/

#include <stdio.h>

struct record {
    int number;
    struct record *next_num;
};

int main(void)
{
    struct record name1, name2, name3;
    struct record *recd_pointer = &name1;
    int sum = 0;
```

Type Specifiers

```
name1.number = 144;
name2.number = 203;
name3.number = 488;

name1.next_num = &name2;
name2.next_num = &name3;
name3.next_num = NULL;

while (recd_pointer != NULL)
{
    sum += recd_pointer->number;
    recd_pointer = recd_pointer->next_num;
}
printf("Sum = %d\n", sum);

return(0);
}
```

The structure type record contains two members: the integer number and next_num, which is a pointer to a structure variable of type record.

The record type variables name1, name2, and name3 are assigned the following values:

Member Name	Value
name1.number	144
name1.next_num	The address of name2
name2.number	203
name2.next_num	The address of name3
name3.number	488
name3.next_num	NULL (Indicating the end of the linked list.)

The variable recd_pointer is a pointer to a structure of type record. It is initialized to the address of name1 (the beginning of the linked list).

The **while** loop causes the linked list to be scanned until recd_pointer equals NULL. The statement:


```
recd_pointer = recd_pointer->next_num;
```

advances the pointer to the next object in the list.

RELATED REFERENCES

- "Chapter 4. Declarators" on page 73
- "Initializers" on page 79
- "Incomplete Types" on page 71
- "Dot Operator ." on page 107
- "Arrow Operator ->" on page 107

Alignment of Structures

 Normal structure alignment aligns the structure members on their natural boundaries and ends the structure on its natural boundary. The alignment of the structure is that of its strictest member. The compiler performs normal alignment when your program meets one of the following conditions:

- It does not specify the #pragma pack directive
- It specifies #pragma pack() before the structure declaration
- It specifies #pragma pack(full) before the structure declaration

To change the alignment back to what it was before the last `#pragma pack`, use the `reset` option.

Consider if, by default, the compiler packs data types along boundaries smaller than those specified by `#pragma pack`. The compiler still aligns them along the smaller boundaries. For example, the compiler always aligns type `char` along a 1-byte boundary, regardless of the value of `#pragma pack`.

Consider when more than one `#pragma pack` directive appears in a structure defined in an inlined function. In that case, the `#pragma pack` directive that is in effect at the beginning of the structure takes precedence.

For information on packing C structures, see “_Packed Qualifier (z/OS C Only)” on page 74. For information on alignment of unions, see “Alignment of Unions” on page 64. For information how to use the `#pragma pack` directive to change alignment, see “pack” on page 249.

Alignment of Nested Structures

z/OS A nested structure has the alignment that precedes its declaration, not the alignment of the structure in which it is contained.

```
#pragma pack ()           // full alignment
    struct nested {
        int x;
        char y;
        int z;
    };

    #pragma pack(1)        // 1-byte alignment
    struct packedcxx{
        char a;
        short b;
        struct nested s1;   // full alignment
    };
```

Unions

C A *union* is an object similar to a structure except that all of its members start at the same location in memory. A union can contain only one of its members at a time. The members of a union can be of any data type.

The storage allocated for a union is the storage required for the largest member of the union (plus any padding that is required so that the union will end at a natural boundary of its strictest member).

C++ In C++, a union is a limited form of the class type. It can contain access specifiers (public, protected, private), member data, and member functions, including constructors and destructors. It cannot contain virtual member functions or static data members. Default access of members in a union is public. A union cannot be used as a base class and cannot be derived from a base class.

C++ A C++ union member cannot be a class object that has a constructor, destructor, or overloaded copy assignment operator. Also, a union cannot have members of reference type. In C++, a member of a union cannot be declared with the keyword **static**.

RELATED REFERENCES

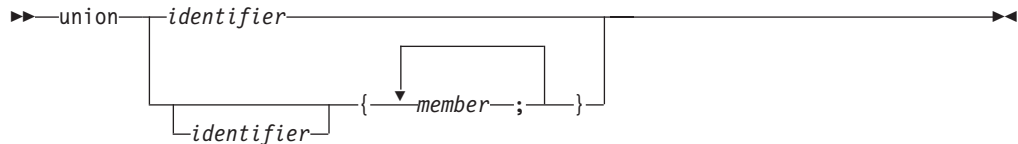
Type Specifiers

- “Member Functions” on page 295
- “Constructors and Destructors Overview” on page 341
- “Virtual Functions” on page 333
- “Overloading Assignments” on page 274
- “static Storage Class Specifier” on page 42

Declaring a Union

C A *union type definition* contains the **union** keyword followed by an identifier (optional) and a brace-enclosed list of members.

A union definition has the following form:



A *union declaration* has the same form as a union definition except that the declaration has no brace-enclosed list of members.

The *identifier* is a tag given to the union specified by the member list. If you specify a tag, any subsequent declaration of the union (in the same scope) can be made by declaring the tag and omitting the member list. If you do not specify a tag, you must put all variable definitions that refer to that union within the statement that defines the data type.

The list of members provides the data type with a description of the objects that can be stored in the union.

A union member definition has same form as a variable declaration.

You can reference one of the possible members of a union the same way as referencing a member of a structure.

For example:

```
union {  
    char birthday[9];  
    int age;  
    float weight;  
} people;  
  
people.birthday[0] = '\n';
```

assigns `'\n'` to the first element in the character array `birthday`, a member of the union `people`.

A union can represent only one of its members at a time. In the example, the union `people` contains either `age`, `birthday`, or `weight` but never more than one of these. The `printf` statement in the following example does not give the correct result because `people.age` replaces the value assigned to `people.birthday` in the first line:

```
#include <stdio.h>  
#include <string.h>
```

```

union {
    char birthday[9];
    int age;
    float weight;
} people;

int main(void) {
    strcpy(people.birthday, "03/06/56");
    printf("%s\n", people.birthday);
    people.age = 38;
    printf("%s\n", people.birthday);
}

```

The output of the above example will be similar to the following:

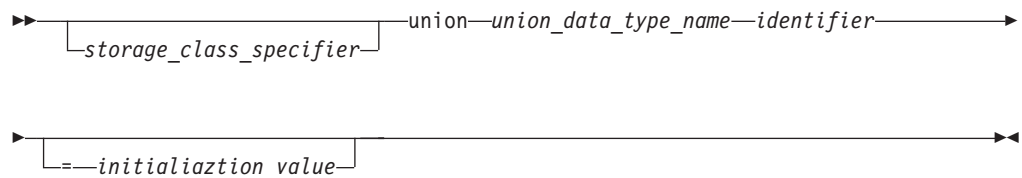
```

03/06/56
&

```

Defining a Union Variable

C A union variable definition has the following form:



You must declare the union data type before you can define a union having that type.

You can define a union data type and a union of that type in the same statement by placing the variable declarator after the data type definition.

You can only initialize the first member of a union.

The following example shows how you would initialize the first union member birthday of the union variable people:

```

union {
    char birthday[9];
    int age;
    float weight;
} people = {"23/07/57"};

```

To define union type and a union variable in one statement, put a declarator after the type definition. The storage class specifier for the variable must go at the beginning of the statement.

RELATED REFERENCES

- “Storage Class Specifiers” on page 34
- “Unions” on page 59
- “Type Specifiers” on page 44
- “volatile and const Qualifiers” on page 69

Defining Packed Unions (z/OS)

C To qualify a C union as packed, use `_Packed`.

Type Specifiers

► C++ z/OS C++ does not support the `_Packed` qualifier. To change the alignment of C++ unions, use the `#pragma pack` directive (which both C and C++ support). For more information on this directive, see “pack” on page 249.

Packed and nonpacked unions cannot be assigned to each other, regardless of their type.

The `#pragma pack` does not affect the memory layout of the union members. Each member starts at offset zero. The `#pragma pack` directive does affect the total alignment restriction of the whole union.

In the following example, each of the elements in the nonpacked `n_array` is of type union `uu`:

```
union uu{
    short    a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu    n_array[2];
/* _Packed union is not supported for C++ */
_Packed union uu    p_array[2];
```

Because it is not packed, each element in the nonpacked `n_array` has an alignment restriction of 2 bytes. (The largest alignment requirement among the union members is that of short `a`.) There is 1 byte of padding at the end of each element to enforce this requirement.

In the packed array, `p_array`, each element is of type `_Packed union uu`. Because every element aligned on the byte boundary, each element has a length of only 3 bytes, instead of the 4 bytes in the previous example.

The following equivalent C++ example uses the `#pragma pack` directive instead of the `_Packed` qualifier:

```
union uu {
    short    a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu    n_array[2];
#pragma pack(pack)
union uu p_array[2];
#pragma pack(reset)
```

Anonymous Unions


An *anonymous union* is a union without a class name. It cannot be followed by a declarator. An anonymous union is not a type; it defines an unnamed object and it cannot have member functions.

► z/OS z/OS C supports anonymous unions only when you use the `LANGlvl(COMMONC)` compiler option.

The member names of an anonymous union must be distinct from other names within the scope in which the union is declared. You can use member names directly in the union scope without any additional member access syntax.

For example, in the following code fragment, you can access the data members `i` and `cptr` directly because they are in the scope containing the anonymous union. Because `i` and `cptr` are union members and have the same address, you should only use one of them at a time. The assignment to the member `cptr` will change the value of the member `i`.

```
void f()
{
    union { int i; char* cptr ; };
    //      .
    //      .
    //      .
    i = 5;
    cptr = "string_in_union"; // overrides the value 5
}
```

 An anonymous union cannot have protected or private members. A global or namespace anonymous union must be declared with the keyword **static**.

RELATED REFERENCES

- “static Storage Class Specifier” on page 42
- “Member Functions” on page 295

Examples of Unions

The following example defines a union data type (not named) and a union variable (named `length`). The member of `length` can be a **long int**, a **float**, or a **double**.

```
union {
    float meters;
    double centimeters;
    long inches;
} length;
```

The following example defines the union type data as containing one member. The member can be named `charctr`, `whole`, or `real`. The second statement defines two data type variables: `input` and `output`.

```
union data {
    char charctr;
    int whole;
    float real;
};
union data input, output;
```

The following statement assigns a character to `input`:

```
input.charctr = 'h';
```

The following statement assigns a floating-point number to member `output`:

```
output.real = 9.2;
```

The following example defines an array of structures named `records`. Each element of `records` contains three members: the integer `id_num`, the integer `type_of_input`, and the union variable `input`. `input` has the union data type defined in the previous example.

Type Specifiers

```
struct {
    int id_num;
    int type_of_input;
    union data input;
} records[10];
```

The following statement assigns a character to the structure member `input` of the first element of **records**:

```
records[0].input.charctr = 'g';
```

RELATED REFERENCES

- “Initializers” on page 79
- “Structures” on page 51
- “Dot Operator `.`” on page 107
- “Arrow Operator `->`” on page 107

Alignment of Unions

z/OS You can perform packing in a union. Each member starts at offset zero, and the entire union spans as many bytes as its largest element. The `#pragma pack` affects the total alignment restriction of the whole union. Consider the following example:

Without Packing:

```
union uu {
    short a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu array[2];
```

First, consider the non-packed array. Each of its elements is of type union `uu`. Since it is non-packed, every element has an alignment restriction of 2 bytes. The largest alignment requirement among the union members is that of short `a`. There is one byte of padding at the end of each element to enforce this requirement.



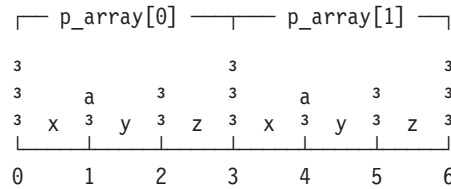
With #pragma pack(packed):

```
#pragma pack(packed)

union uu {
    short a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu p1_array[2];
```

Now consider the packed array `p1_array`. Since the example specifies `#pragma pack(packed)`, the alignment restriction of every element is the byte boundary. Therefore, each element has a length of only 3 bytes, as opposed to the 4 bytes of the previous case.



For information on structure alignment, see “Alignment of Structures” on page 58 and “Alignment of Nested Structures” on page 59. For information how to use the `#pragma pack` directive to change alignment, see “pack” on page 249.

Examples:

In a header file, file.h:

```
#pragma pack(packed)

struct jeff{                /* this structure is packed */
    float bill;             /*    along 1-byte boundaries    */
    int *chris;
}

#pragma pack(reset)        /* reset to previous alignment rule*/

:
:
```

In a source file, file.cxx:

```
#pragma pack(full)

#include "file.h"           // inside the header file,
                           // the alignment rule is set to 1-byte
                           // and then reset to the system default

struct dor{                // this structure is packed
    double stephen;         // using the system default alignment
    long alex;
}
```

Enumerations

An *enumeration* data type represents a set of values that you declare. You can define an enumeration data type and all variables that have that enumeration type in one statement, or you can declare an enumeration type separately from the definition of variables of that type. The identifier associated with the data type (not an object) is called an *enumeration tag*.

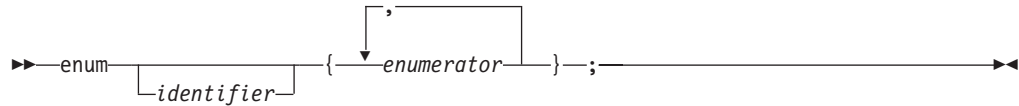
RELATED REFERENCES

- “Type Specifiers” on page 44

Declaring an Enumeration Data Type

An enumeration type declaration contains the **enum** keyword followed by an optional identifier (the enumeration tag) and a brace-enclosed list of enumerators. Commas separate each enumerator in the enumerator list.

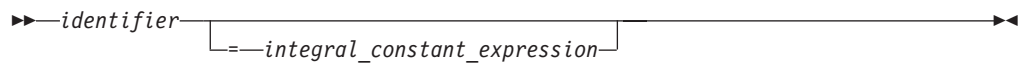
Type Specifiers



The keyword **enum**, followed by the identifier, names the data type (like the tag on a **struct** data type). The list of enumerators provides the data type with a set of values.

► C++ ► C In C, each enumerator represents an integer value. In C++, each enumerator represents a value that can be converted to an integral value.

An enumerator has the form:



To conserve space, enumerations may be stored in spaces smaller than that of an `int`.

Enumeration Constants

When you define an enumeration data type, you specify a set of identifiers that the data type represents. Each identifier in this set is an *enumeration constant*.

The value of the constant is determined in the following way:

1. An equal sign (=) and a constant expression after the enumeration constant gives an explicit value to the constant. The identifier represents the value of the constant expression.
2. If no explicit value is assigned, the leftmost constant in the list receives the value zero (0).
3. Identifiers with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous identifier.

► C In C, enumeration constants have type **int**.

► C++ In C++, each enumeration constant has a value that can be promoted to a signed or unsigned integer value and a distinct type that does not have to be integral. Use an enumeration constant anywhere an integer constant is allowed, or for C++, anywhere a value of the enumeration type is allowed.

Each enumeration constant must be unique within the scope in which the enumeration is defined. In the following example, second declarations of `average` and `poor` cause compiler errors:

```
func()
{
    enum score { poor, average, good };
    enum rating { below, average, above };
    int poor;
}
```

The following data type declarations list oats, wheat, barley, corn, and rice as enumeration constants. The number under each constant shows the integer value.

```
enum grain { oats, wheat, barley, corn, rice };
/*      0      1      2      3      4      */

enum grain { oats=1, wheat, barley, corn, rice };
/*      1      2      3      4      5      */

enum grain { oats, wheat=10, barley, corn=20, rice };
/*      0      10      11      20      21      */
```

It is possible to associate the same integer with two different enumeration constants. For example, the following definition is valid. The identifiers `suspend` and `hold` have the same integer value.

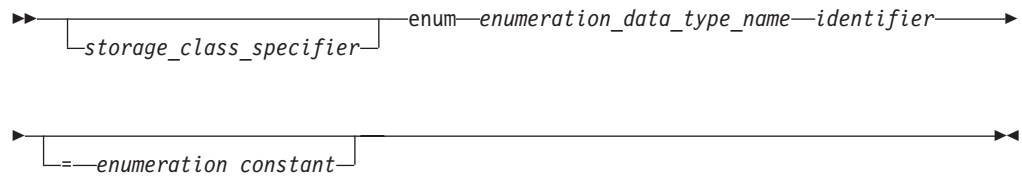
```
enum status { run, clear=5, suspend, resume, hold=6 };
/*      0      5      6      7      6      */
```

RELATED REFERENCES

- “Integer Variables” on page 49
- “Integral and Floating-Point Promotions” on page 143

Defining Enumeration Variables

An enumeration variable definition has the following form:



You must declare the enumeration data type before you can define a variable having that type.

► **C++** The initializer for an enumeration variable contains the `=` symbol followed by an expression *enumeration_constant*. In C++, the initializer must have the same type as the associated enumeration type.

The first line of the following example declares the enumeration `grain`. The second line defines the variable `g_food` and gives `g_food` the initial value of `barley` (2).

```
enum grain { oats, wheat, barley, corn, rice };
enum grain g_food = barley;
```

The type specifier `enum grain` indicates that the value of `g_food` is a member of the enumerated data type `grain`.

► **C++** In C++, the **enum** keyword is optional when declaring a variable with enumeration type. However, it is required when declaring the enumeration itself. For example, both statements declare a variable of enumeration type:

```
enum grain g_food = barley;
grain cob_food = corn;
```

RELATED REFERENCES

- “Storage Class Specifiers” on page 34

Type Specifiers

Defining an Enumeration Type and Enumeration Objects

You can define a type and a variable in one statement by using a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, you must put the storage class specifier at the beginning of the declaration. For example:

```
register enum score { poor=1, average, good } rating = good;
```

C++ C++ also lets you put the storage class immediately before the declarator list. For example:

```
enum score { poor=1, average, good } register rating = good;
```

Either of these examples is equivalent to the following two declarations:

```
enum score { poor=1, average, good };
register enum score rating = good;
```

Both examples define the enumeration data type `score` and the variable `rating`. `rating` has the storage class specifier **register**, the data type `enum score`, and the initial value `good`.

Combining a data type definition with the definitions of all variables having that data type lets you leave the data type unnamed. For example:

```
enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
      Saturday } weekday;
```

defines the variable `weekday`, which can be assigned any of the specified enumeration constants.

Example Program Using Enumerations

The following program receives an integer as input. The output is a sentence that gives the French name for the weekday that is associated with the integer. If the integer is not associated with a weekday, the program prints "C'est le mauvais jour."

CCNRAAN

```
/**
 ** Example program using enumerations
 **/

#include <stdio.h>

enum days {
    Monday=1, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday
} weekday;

void french(enum days);

int main(void)
{
    int num;

    printf("Enter an integer for the day of the week.  "
           "Mon=1,...,Sun=7\n");
    scanf("%d", &num);
    weekday=num;
    french(weekday);
    return(0);
}
```

```

void french(enum days weekday)
{
    switch (weekday)
    {
        case Monday:
            printf("Le jour de la semaine est lundi.\n");
            break;
        case Tuesday:
            printf("Le jour de la semaine est mardi.\n");
            break;
        case Wednesday:
            printf("Le jour de la semaine est mercredi.\n");
            break;
        case Thursday:
            printf("Le jour de la semaine est jeudi.\n");
            break;
        case Friday:
            printf("Le jour de la semaine est vendredi.\n");
            break;
        case Saturday:
            printf("Le jour de la semaine est samedi.\n");
            break;
        case Sunday:
            printf("Le jour de la semaine est dimanche.\n");
            break;
        default:
            printf("C'est le mauvais jour.\n");
    }
}

```

volatile and const Qualifiers

The **volatile** qualifier maintains consistency of memory access to data objects.

The **volatile** qualifier is useful for data objects having values that may be changed in ways unknown to your program (such as the system clock). Portions of an expression that reference **volatile** objects are not to be changed or removed.

The **const** qualifier explicitly declares a data object as a data item that cannot be changed. Its value is set at initialization. You cannot use **const** data objects in expressions requiring a modifiable lvalue. For example, a **const** data object cannot appear on the left-hand side of an assignment statement.

These type qualifiers are only meaningful in expressions that are lvalues.

For a **volatile** or **const** pointer, you must put the keyword between the * and the identifier. For example:

```

int * volatile x;      /* x is a volatile pointer to an int */
int * const y = &z;    /* y is a const pointer to the int variable z */

```

For a pointer to a **volatile** or **const** data object, the type specifier, qualifier, and storage class specifier can be in any order. For example:

```

volatile int *x;      /* x is a pointer to a volatile int */
or
int volatile *x;      /* x is a pointer to a volatile int */

const int *y;         /* y is a pointer to a const int */
or
int const *y;         /* y is a pointer to a const int */

```

In the following example, the pointer to y is a constant. You can change the value that y points to, but you cannot change the value of y:

Type Specifiers

```
int * const y
```

In the following example, the value that `y` points to is a constant integer and cannot be changed. However, you can change the value of `y`:

```
const int * y
```

For other types of **volatile** and **const** variables, the position of the keyword within the definition (or declaration) is less important. For example:

```
volatile struct omega {  
    int limit;  
    char code;  
} group;
```

provides the same storage as:

```
struct omega {  
    int limit;  
    char code;  
} volatile group;
```

In both examples, only the structure variable `group` receives the **volatile** qualifier. Similarly, if you specified the **const** keyword instead of **volatile**, only the structure variable `group` receives the **const** qualifier. The **const** and **volatile** qualifiers when applied to a structure, union, or class also apply to the members of the structure, union, or class.

Although enumeration, class, structure, and union variables can receive the **volatile** or **const** qualifier, enumeration, class, structure, and union tags do not carry the **volatile** or **const** qualifier. For example, the `blue` structure does not carry the **volatile** qualifier:

```
volatile struct whale {  
    int weight;  
    char name[8];  
} beluga;  
  
struct whale blue;
```

The keywords **volatile** and **const** cannot separate the keywords **enum**, **class**, **struct**, and **union** from their tags.

You can declare or define a **volatile** or **const** function only if it is a C++ member function. You can define or declare any function to return a pointer to a **volatile** or **const** function.

You can put more than one qualifier on a declaration but you cannot specify the same qualifier more than once on a declaration.



An item can be both **const** and **volatile**. In this case the item cannot be legitimately modified by its own program but can be modified by some asynchronous process.

RELATED REFERENCES

- “Lvalues and Rvalues” on page 99
- “Pointers” on page 81
- “Chapter 12. Classes” on page 283
- “Structures” on page 51
- “Unions” on page 59

Incomplete Types

The following are incomplete types:

- Type **void**
- Array of unknown size
- Structure, union, or enumerations that have no definition
-  C++ Pointers to class types that are declared but not defined
-  C++ Classes that are declared but not defined

The following example is an incomplete type:

```
void *incomplete_ptr;
```

```
struct dimension linear; /* no previous definition of dimension */
```

void is an incomplete type that cannot be completed. Incomplete structure or union and enumeration tags must be completed before being used to declare an object, although you can define a pointer to an incomplete structure or union.

RELATED REFERENCES

- “void Type” on page 50
- “Arrays” on page 86
- “Structures” on page 51
- “Unions” on page 59
- “Incomplete Class Declarations” on page 288

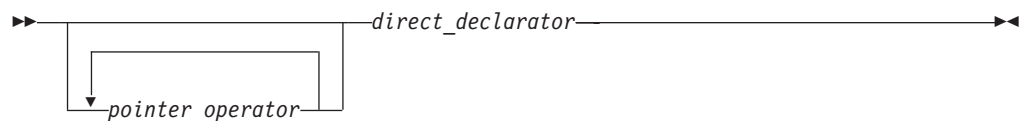
Chapter 4. Declarators

A *declarator* designates a data object or function. Declarators appear in most data definitions and declarations and in some type definitions.

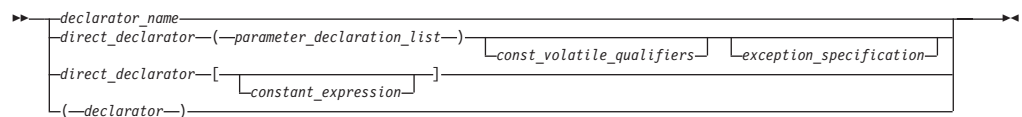
In a declarator, you can specify the type of an object to be an array, a pointer, or a reference. You can also perform initialization in a declarator.

A declarator has the form:

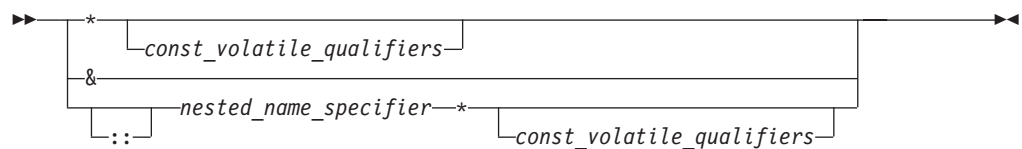
declarator



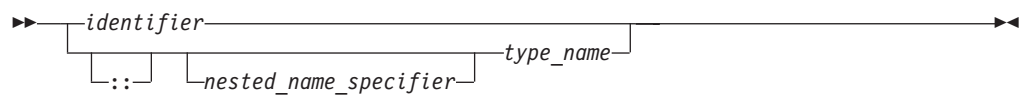
direct_declarator



pointer_operator



declarator_name



C++ The following variables or delimiters are only available in C++:

- *exception_specification*
- *nested_name_specifier*
- `::` (scope resolution operator)

The *const_volatile_qualifiers* variable represents one or a combination of **const** and **volatile**.

C **C++** In C, you cannot declare or define a **volatile** or **const** function. C++ class member functions can be qualified with **const** or **volatile**.

Declarators

z/OS The z/OS C compiler implements the `_Packed` qualifier, and the z/OS C++ compiler implements the `_Export` qualifier.

The following table illustrate some examples of declarators:

Example	Description
<code>int owner</code>	owner is an int data object.
<code>int *node</code>	node is a pointer to an int data object.
<code>int names[126]</code>	names is an array of 126 int elements.
<code>int *action()</code>	action is a function returning a pointer to an int .
<code>volatile int min</code>	min is an int that has the volatile qualifier.
<code>int * volatile volume</code>	volume is a volatile pointer to an int .
<code>volatile int * next</code>	next is a pointer to a volatile int .
<code>volatile int * sequence[5]</code>	sequence is an array of five pointers to volatile int objects.
<code>extern const volatile int op_system_clock</code>	op_system_clock is a constant and volatile integer with static storage duration and external linkage.
z/OS <code>_Packed struct struct_type s</code>	s is a packed structure of type struct_type.

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69
- “Enumerations” on page 65
- “Pointers” on page 81
- “Arrays” on page 86
- “Structures” on page 51
- “Unions” on page 59
- “Exception Specifications” on page 412
- “Scope of Class Names” on page 287

`_Packed` Qualifier (z/OS C Only)

z/OS z/OS C stores data elements of structure and unions in memory on an address boundary specific for that data type. For example, a `double` value is stored in memory on a doubleword (8-byte) boundary. There may be gaps left in memory between structure and union elements to align elements on their natural boundaries. You can reduce the padding of bytes within a structure or union by *packing*.

The `_Packed` qualifier removes padding between members of structures and affects the alignment of unions whenever possible. However, the storage that is saved using packed structures and unions may come at the expense of run-time performance. Most machines access data more efficiently if the data aligns on appropriate boundaries. With packed structures and unions, members are generally not aligned on natural boundaries. The result is that operations using the class-member access operators (`.` and `->`) are slower.

Note: z/OS C/C++ aligns pointers on their natural boundaries, 4 bytes, even in packed structures and unions.

You can only use `_Packed` with structures or unions. If you use `_Packed` with other types, z/OS C generates a warning message, and the qualifier has no effect on the declarator it qualifies. Packed and nonpacked structures and unions have different storage layouts.

You cannot perform comparisons between packed and nonpacked structures, or unions of the same type. Packed and nonpacked structures or unions cannot be assigned to each other, regardless of their type.

You cannot pass a packed union or packed structure as a function parameter if the function expects a nonpacked version. If the function expects a packed structure or a packed union, you cannot pass a nonpacked version as a function parameter.

If you specify the `_Packed` qualifier on a structure or union that contains a structure or union as a member, the qualifier is not passed on to the contained structure or union. See “Pragma Directives (#pragma)” on page 219 for more information on `#pragma pack`.

`__cdecl` Keyword (z/OS C++ Only)

z/OS Use the `__cdecl` keyword to set linkage conventions for function calls in C++ applications. You can use the `__cdecl` linkage keyword at any language level. The `__cdecl` keyword instructs the compiler to read and write a parameter list by using C linkage conventions.

To set the `__cdecl` calling convention for a function, place the linkage keyword immediately before the function name or at the beginning of the declarator. For example:

```
void __cdecl f();
char (__cdecl *fp) (void);
```

z/OS C++ allows the `__cdecl` keyword on member functions and nonmember functions. These functions can be static or non-static. It also allows the keyword on pointer-to-member function types and the typedef specifier.

Note: The compiler accepts both `_cdecl` and `__cdecl` (both single and double underscore).

Following is an example:

```
// C++ nonmember functions
void __cdecl f1();
static void __cdecl f2();

// pointer to member function type
char (__cdecl *A::mfp) (void);

// typedef
typedef void (*_cdecl void_fcn)(int);
// C++ member functions
class A {
public:
    void __cdecl func();
    static void __cdecl func1();
}

// Template member functions
template <class T> X {
public:
```

Declarators

```
void __cdecl func();
static void __cdecl func1();
}

// Template functions
template <class T> T __cdecl foo(T i) {return i+1;}
template <class T> T static __cdecl foo2(T i) {return i+1;}
```

Semantics of __cdecl

z/OS The __cdecl linkage keyword only affects parameter passing; it does not prevent function name mangling. Therefore, you can still overload functions with non-default linkage. Note that you only acquire linkage by explicitly using the __cdecl keyword. It overrides the linkage that it inherits from an extern "linkage" specification.

Following is an example:

```
void __cdecl foo(int); // C linkage with name mangled
void __cdecl foo(char) // overload foo() with char is OK

void foo(int(*)());
// overload on linkage of function
void foo(int (__cdecl *)());
//pointer parameter is OK

extern "C++" {
    void __cdecl foo(int);
    // foo() has C linkage with name mangled
}

extern "C" {
    void __cdecl foo(int);
    // foo() has C linkage with name mangled
}
```

If the function is redeclared, the linkage keyword must appear in the first declaration, otherwise z/OS issues an error diagnostic. Following are two examples:

```
int c_cf();
int __cdecl c_cf();
// Error 1251. The previous declaration did not have a linkage
specification

int __cdecl c_cf();
int c_cf();
// OK, the linkage is inherited from the first declaration
```

Examples of __cdecl Use

z/OS Prior to the Version 2 Release 4 OS/390 C++ compiler, the C++ function pointer could not pass in the C function parameter list as the compiler did not support __cdecl linkage. The following examples illustrate how you can pass in the C parameter list by using the __cdecl linkage:

Example 1

```
/*-----*/
/* C++ source file */
/*-----*/
//
// C++ Application: passing a C++ function pointer to a C function
//
#include <stdio.h>

// C++ function declares with C calling convention
```

```

void __cdecl callcxx() {
    printf(" I am a C++ function\n");
}

// declare a function pointer with __cdecl linkage
void (__cdecl *p1)();

// declare an extern C function,
// accepting a __cdecl function pointer
extern "C" {
    void CALLC(void (__cdecl *pp)());
}

// assign the function pointer to a __cdecl function
int main() {
    p1 = callcxx;

// call the C function with the __cdecl function pointer
    CALLC(p1);
}

```

Example 2

```

/*-----*/
/* C source file                                */
/*-----*/

/*                                          */
/* C Routine:  receiving a function pointer with C linkage      */
/*                                          */
#include <stdio.h>
extern void CALLC(void (*pp)()){
    printf(" I am a C function\n");
    (*pp)();  // call the function passed in
}

```

_Export Keyword (z/OS C++ only)

z/OS Use the `_Export` keyword (in C++ applications only) with a function name or external variable to declare that it is to be exported (made available to other modules). For example:

```
int _Export anthony(float);
```

The above statement exports the function `anthony`, if you define the function within the compilation unit. You must define the function in the same compilation unit in which you use the `_Export` keyword.

z/OS C/C++ allows `_Export` only at file scope. You cannot use it in a typedef. You cannot apply the `_Export` keyword to the return type of a function. For example, the following declaration causes an error :

```
_Export int * a(); // error
```

The following declaration, however, would export `a()`.

```
int * _Export a(); // error
```

The `_Export` keyword must immediately precede the function or object name. If the `_Export` keyword is repeated in a declaration, z/OS C++ issues a warning when you specify the `info(gen)` option.

Since `_Export` is part of the declarator, it affects only the closest identifier. In the following declaration, `_Export` only modifies `a`:

Declarators

```
int _Export a, b;
```

You can use `_Export` at any language level.

The `_Export` keyword is an alternative to the `#pragma export()` directive. Note however that their semantics are not equivalent. The `_Export` keyword can apply to a class tag, object, or function and is generally easier to use. The `#pragma export()` directive only applies to an object or function. If you attempted to export an overloaded function or a class member function using the `#pragma export()` directive as follows, you would receive a syntax error.

```
#pragma export(f) // syntax error since f has not been declared
void f();
void f(int);
class C1 { void f(); void f(int); };
class C2 { void f(); void f(int); class C3 { void f(); }; };
```


In addition, you need to specify the exact identifier that you want exported. For example, `#pragma export(f(int))` would export `void f(int)` for the above example.

To export member functions, you may apply the `_Export` keyword to the function declaration, but the function definition must not be inlined. For example:

```
class X {
    public:
        ...
        void _Export Print();
        ...
};

void X::Print() {
    ...
}
```

The above example will cause the function `X::Print()` to be exported.

 It is not possible to export C++ inlined functions even with the `#pragma export()` directive.

If the you apply the `_Export` keyword to a class, then z/OS C/C++ automatically exports any static members of that class. The `_Export` keyword exports the *public interface* of a class, so you should export all public methods and data members using `_Export`. In the example below, both `X::Print()` and `X::GetNext()` will be exported.

```
class _Export X {
    public:
        ...
        void static Print();
        int GetNext();
        ...
};

void X:: static Print() {
    ...
}
int X::GetNext() {
    ...
}
```


The above examples demonstrate that you can either export specific members of a class or the entire class itself. Note that the `_Export` keyword can be applied to class tags in nested class declarations.

The function `main()` cannot be exported. For a description of `#pragma export`, see “export” on page 232.

Initializers

An *initializer* is an optional part of a data declaration that specifies an initial value of a data object.

The initialization properties of each data type are described in the section for that data type.

The initializer consists of the `=` symbol followed by an initial *expression* or a braced list of initial expressions separated by commas. The number of initializers must not be more than the number of elements to be initialized. The initial expression evaluates to the first value of the data object.

To assign a value to an arithmetic or pointer type, use the simple initializer: `= expression`. For example, the following data definition uses the initializer `= 3` to set the initial value of `group` to 3:

```
int group = 3;
```

For unions, structures, and aggregate classes (classes with no constructors, base classes, virtual functions, or private or protected members), the set of initial expressions must be enclosed in `{ }` (braces) unless the initializer is a string literal.

If the initializer of a character string is a string literal, the `{ }` are optional. Individual expressions must be separated by commas, and groups of expressions can be enclosed in braces and separated by commas.

In an array, structure, or union initialized using a brace-enclosed initializer list, any members or subscripts that are not initialized are implicitly initialized to zero of the appropriate type.

C++ An initializer of the form *(expression)* can be used to initialize fundamental types in C++. For example, the following two initializations are identical:

```
int group = 3;
int group(3);
```

You can also use the *(expression)* form to initialize C++ classes. However, the form with parentheses and the form with the assignment operator are not identical. The form with parenthesis calls the copy constructor of the class. The form with the assignment operator calls the copy assignment operator of the class.

C **C++** You can initialize variables at namespace scope with nonconstant expressions. In C, you cannot do the same at global scope.

If your code jumps over declarations that contain initializations, the compiler generates an error. For example, the following code is not valid:

Initializers

```
goto skiplabel;    // error - jumped over declaration
int i = 3;         // and initialization of i

skiplabel: i = 4;
```

You can initialize classes in external, static, and automatic definitions. The initializer contains an = (equal sign) followed by a brace-enclosed, comma-separated list of values. You do not need to initialize all members of a class.

In the following example, only the first eight elements of the array `grid` are explicitly initialized. The remaining four elements that are not explicitly initialized are initialized as if they were explicitly initialized to zero.

```
static short grid[3] [4] = {0, 0, 0, 1, 0, 0, 1, 1};
```

The initial values of `grid` are:

Element	Value	Element	Value
grid[0] [0]	0	grid[1] [2]	1
grid[0] [1]	0	grid[1] [3]	1
grid[0] [2]	0	grid[2] [0]	0
grid[0] [3]	1	grid[2] [1]	0
grid[1] [0]	0	grid[2] [2]	0
grid[1] [1]	0	grid[2] [3]	0

RELATED REFERENCES

- “Copy Assignment Operators” on page 363
- “Assignment Expressions” on page 134
- “Arrays” on page 86
- “char and wchar_t Type Specifiers” on page 45
- “Simple Type Specifiers” on page 44
- “Enumerations” on page 65
- “Floating-Point Variables” on page 47
- “Integer Variables” on page 49
- “Pointers” on page 81
- “Structures” on page 51
- “Unions” on page 59
- “Storage Class Specifiers” on page 34

C/C++ Data Mapping (z/OS)

► **z/OS** The S/390 architecture has the following boundaries in its memory mapping:

- Byte
- Halfword
- Fullword
- Doubleword

The code that is produced by the C/C++ compiler places data types on natural boundaries. Some examples are:

- Byte boundary for `char`
- Byte boundary for `decimal(n,p)` (C only)
- Halfword boundary for `short int`
- Fullword boundary for `int`
- Fullword boundary for `long int`
- Fullword boundary for pointers
- Fullword boundary for `float`
- Doubleword boundary for `double`
- Doubleword boundary for `long double`

For each external defined variable, the z/OS C/C++ compiler defines a writeable static data instance of the same name. The compiler places other external variables, such as those in programs that you compiled with the NORENT compiler option, in separate CSECTs that are based on their names.

Pointers

A *pointer* type variable holds the address of a data object or a function. A pointer can refer to an object of any one data type except to a bit field or a reference. Some common uses for pointers are:

- To access dynamic data structures such as linked lists, trees, and queues.
- To access elements of an array or members of a structure or C++ class.
- To access an array of characters as a string.
- To pass the address of a variable to a function. (In C++, you can also use a reference to do this.) By referencing a variable through its address, a function can change the contents of that variable.

RELATED REFERENCES

- “Calling Functions and Passing Arguments” on page 164
- “References” on page 92

Declaring Pointers

The following example declares `pcoat` as a pointer to an object having type **long**:

```
extern long *pcoat;
```

If the keyword **volatile** appears before the `*`, the declarator describes a pointer to a **volatile** object. If the keyword **volatile** comes between the `*` and the identifier, the declarator describes a **volatile** pointer. The keyword **const** operates in the same manner as the **volatile** keyword described. In the following example, `pvolt` is a constant pointer to an object having type **short**:

```
short * const pvolt;
```

The following example declares `pnut` as a pointer to an **int** object having the **volatile** qualifier:

```
extern int volatile *pnut;
```

The following example defines `psoup` as a **volatile** pointer to an object having type **float**:

```
float * volatile psoup;
```

The following example defines `pfowl` as a pointer to an enumeration object of type `bird`:

C/C++ Data Mapping

```
enum bird *pfowl;
```

The next example declares `pvish` as a pointer to a function that takes no parameters and returns a **char** object:

```
char (*pvish)(void);
```

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69

Assigning Pointers

When you use pointers in an assignment operation, you must ensure that the types of the pointers in the operation are compatible.

The following example shows compatible declarations for the assignment operation:

```
float subtotal;  
float * sub_ptr;  
// ...  
sub_ptr = &subtotal;  
printf("The subtotal is %f\n", *sub_ptr);
```

The next example shows incompatible declarations for the assignment operation:

```
double league;  
int * minor;  
// ...  
minor = &league;    /* error */
```

RELATED REFERENCES

- “Pointers” on page 81
- “Assignment Expressions” on page 134

Initializing Pointers

The initializer is an `=` (equal sign) followed by the expression that represents the address that the pointer is to contain. The following example defines the variables `time` and `speed` as having type **double** and `amount` as having type pointer to a **double**. The pointer `amount` is initialized to point to `total`:

```
double total, speed, *amount = &total;
```

The compiler converts an unsubscripted array name to a pointer to the first element in the array. You can assign the address of the first element of an array to a pointer by specifying the name of the array. The following two sets of definitions are equivalent. Both define the pointer `student` and initialize `student` to the address of the first element in `section`:

```
int section[80];  
int *student = section;
```

is equivalent to:

```
int section[80];  
int *student = &section[0];
```

You can assign the address of the first character in a string constant to a pointer by specifying the string constant in the initializer.

The following example defines the pointer variable `string` and the string constant `"abcd"`. The pointer `string` is initialized to point to the character `a` in the string `"abcd"`.

```
char *string = "abcd";
```

The following example defines weekdays as an array of pointers to string constants. Each element points to a different string. The pointer weekdays[2], for example, points to the string "Tuesday".

```
static char *weekdays[ ] =
{
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};
```

A pointer can also be initialized to NULL using any integer constant expression that evaluates to 0, for example char * a=0;. Such a pointer is a *NULL pointer*. It does not point to any object.

RELATED REFERENCES

- “Initializers” on page 79
- “Arrays” on page 86

Restrictions on Pointers

C You cannot use pointers to reference bit fields or objects having the **register** storage class specifier.

RELATED REFERENCES

- “register Storage Class Specifier” on page 41
- “Restrictions on z/OS C Pointers”

Restrictions on z/OS C Pointers

z/OS The z/OS C compiler supports only the pointers that are obtained in one of the following ways:

- Directly from a malloc/calloc/realloc call
- As an address of a data type (that is, &variable)
- From constants that refer to valid addresses or from the NULL constant
- Received as a parameter from another C function
- Directly from a call to a service in the z/OS Language Environment that allocates storage, such as CEEGTST

Any bitwise manipulation of a pointer can result in undefined behavior.

Note: See *z/OS C/C++ Programming Guide* for details about receiving the parameter list (argv) in C main, information on preparing your main routine to receive parameters, and on C and C++ parameter passing considerations.

You cannot use pointers to reference bit fields or objects that have the **register** storage class specifier.

Packed and nonpacked objects have different memory layouts. Consequently, a pointer to a packed structure or union is incompatible with a pointer to a corresponding nonpacked structure or union. As a result, comparisons and assignments between pointers to packed and nonpacked objects are not valid.

You can, however, perform these assignments and comparisons with type casts. In the following example, the cast operation lets you compare the two pointers, but you must be aware that ps1 still points to a nonpacked object:

C/C++ Data Mapping

```
int main(void)
{
    _Packed struct ss *ps1;
    struct ss          *ps2;

    .
    .
    .
    ps1 = (_Packed struct ss *)ps2;
    .
    .
    .
}
```

Using Pointers

Two operators are commonly used in working with pointers, the address (&) operator and the indirection (*) operator. You can use the & operator to refer to the address of an object. For example, the assignment in the following function assigns the address of `x` to the variable `p_to_int`. The variable `p_to_int` has been defined as a pointer:

```
void f(int x, int *p_to_int)
{
    p_to_int = &x;
}
```

The * (indirection) operator lets you access the value of the object a pointer refers to. The assignment in the following example assigns to `y` the value of the object that `p_to_float` points to:

```
void g(float y, float *p_to_float) {
    y = *p_to_float;
}
```

The assignment in the following example assigns the value of `z` to the variable that `*p_to_z` references:

```
void h(char z, char *p_to_char) {
    *p_to_char = z;
}
```

RELATED REFERENCES

- “Address &” on page 116
- “Indirection *” on page 117

Pointer Arithmetic

You can perform a limited number of arithmetic operations on pointers. These operations are:

- Increment and decrement
- Addition and subtraction
- Comparison
- Assignment

The increment (++) operator increases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the ++ makes the pointer refer to the third element in the array.

The decrement (--) operator decreases the value of a pointer by the size of the data object the pointer refers to. For example, if the pointer refers to the second element in an array, the -- makes the pointer refer to the first element in the array.

You can add an integer to a pointer but you cannot add a pointer to a pointer.

If the pointer `p` points to the first element in an array, the following expression causes the pointer to point to the third element in the same array:

```
p = p + 2;
```

If you have two pointers that point to the same array, you can subtract one pointer from the other. This operation yields the number of elements in the array that separate the two addresses that the pointers refer to.

You can compare two pointers with the following operators: `==`, `!=`, `<`, `>`, `<=`, and `>=`.

Pointer comparisons are defined only when the pointers point to elements of the same array. Pointer comparisons using the `==` and `!=` operators can be performed even when the pointers point to elements of different arrays.

You can assign to a pointer the address of a data object, the value of another compatible pointer or the `NULL` pointer.

RELATED REFERENCES

- “Pointers” on page 81
- “Increment ++” on page 114
- “Arrays” on page 86
- “Decrement —” on page 114
- “Chapter 5. Expressions and Operators” on page 95

Example Program Using Pointers

The following program contains pointer arrays:

CNNRAAQ

```

/*****
**   Program to search for the first occurrence of a specified
**   character string in an array of character strings.
*****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 20

int main(void)
{
    static char *names[ ] = { "Jim", "Amy", "Mark", "Sue", NULL };
    char * find_name(char **, char *);
    char new_name[SIZE], *name_pointer;

    printf("Enter name to be searched.\n");
    scanf("%s", new_name);
    name_pointer = find_name(names, new_name);
    printf("name %s%sfound\n", new_name,
        (name_pointer == NULL) ? " not " : " ");
} /* End of main */

/*****
**   Function find_name. This function searches an array of
**   names to see if a given name already exists in the array.
**   It returns a pointer to the name or NULL if the name is
*****/
```

C/C++ Data Mapping

```
**      not found.                                **
**                                                    **
** char **array is a pointer to arrays of pointers (existing names) **
** char *strng is a pointer to character array entered (new name) **
**                                                    **
*****/

char * find_name(char **array, char *strng)
{
    for (; *array != NULL; array++)                /* for each name      */
    {
        if (strcmp(*array, strng) == 0)            /* if strings match      */
            return(*array);                        /* found it!             */
    }
    return(*array);                                /* return the pointer     */
} /* End of find_name */
```

Interaction with this program could produce the following sessions:

Output Enter name to be searched.

Input Mark

Output name Mark found

or:

Output Enter name to be searched.

Input Deborah

Output name Deborah not found

RELATED REFERENCES

- “Chapter 4. Declarators” on page 73
- “volatile and const Qualifiers” on page 69
- “Initializers” on page 79
- “Address &” on page 116
- “Indirection *” on page 117

Arrays

An *array* is an ordered group of data objects. Each object is called an *element*. All elements within an array have the same data type.

Array elements cannot be of function data type or, in C++, of reference data type. You can, however, declare an array of pointers to functions.

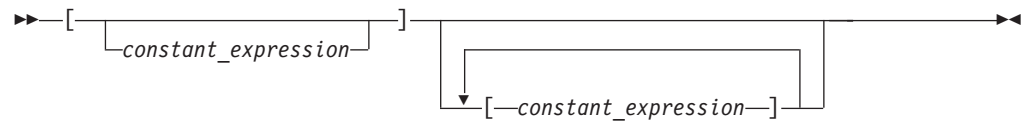
RELATED REFERENCES

- “Declaring Arrays”
- “Initializing Arrays” on page 88

Declaring Arrays

The array declarator contains an identifier followed by an optional *subscript declarator*. An identifier preceded by an * (asterisk) is an array of pointers.

A subscript declarator has the form:



The subscript declarator describes the number of dimensions in the array and the number of elements in each dimension. Each bracketed expression, or subscript, describes a different dimension and must be a constant expression.

The following example defines a one-dimensional array that contains four elements having type **char**:

```
char
list[4];
```

The first subscript of each dimension is 0. The array `list` contains the elements:

```
list[0]
list[1]
list[2]
list[3]
```

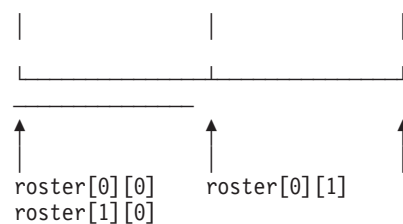
The following example defines a two-dimensional array that contains six elements of type **int**:

```
int
roster[3][2];
```

Multidimensional arrays are stored in row-major order. When elements are referred to in order of increasing storage location, the last subscript varies the fastest. For example, the elements of array `roster` are stored in the order:

```
roster[0][0]
roster[0][1]
roster[1][0]
roster[1][1]
roster[2][0]
roster[2][1]
```

In storage, the elements of `roster` would be stored as:



You can leave the first (and only the first) set of subscript brackets empty in

- Array definitions that contain initializations
- **extern** declarations
- Parameter declarations

In array definitions that leave the first set of subscript brackets empty, the initializer determines the number of elements in the first dimension. In a one-dimensional array, the number of initialized elements becomes the total number of elements. In a multidimensional array, the initializer is compared to the subscript declarator to determine the number of elements in the first dimension.

C/C++ Data Mapping

An unsubscripted array name (for example, `region` instead of `region[4]`) represents a pointer whose value is the address of the first element of the array, provided the array has previously been declared.

Whenever an array is used in a context (such as a parameter) where it cannot be used as an array, the identifier is treated as a pointer. The two exceptions are when an array is used as an operand of the **sizeof** or the address (&) operator.

RELATED REFERENCES

- “extern Storage Class Specifier” on page 37
- “Function Declarations” on page 154
- “sizeof (Size of an Object)” on page 117
- “Address &” on page 116

Initializing Arrays

The initializer for an array contains the = symbol followed by a comma-separated list of constant expressions enclosed in braces ({ }). You do not need to initialize all elements in an array. Elements that are not initialized (in **extern** and **static** definitions only) receive the value 0 of the appropriate type.

Note: Array initializations can be either *fully braced* (with braces around each dimension) or *unbraced* (with only one set of braces enclosing the entire set of initializers).

C++ Avoid placing braces around some dimensions and not around others.

The following definition shows a completely initialized one-dimensional array:

```
static int number[3] = { 5, 7, 2 };
```

The array `number` contains the following values:

Element	Value
<code>number[0]</code>	5
<code>number[1]</code>	7
<code>number[2]</code>	2

The following definition shows a partially initialized one-dimensional array:

```
static int number1[3] = { 5, 7 };
```

The values of `number1` are:

Element	Value
<code>number1[0]</code>	5
<code>number1[1]</code>	7
<code>number1[2]</code>	0

Instead of an expression in the subscript declarator defining the number of elements, the following one-dimensional array definition defines one element for each initializer specified:

```
static int  
item[ ] = { 1, 2, 3, 4, 5 };
```

The compiler gives `item` the five initialized elements:

Element	Value
---------	-------

```

item[0]      1
item[1]      2
item[2]      3
item[3]      4
item[4]      5

```

You can initialize a one-dimensional character array by specifying:

- A brace-enclosed comma-separated list of constants, each of which can be contained in a character
- A string constant (Braces surrounding the constant are optional)

Initializing a string constant places the null character (`\0`) at the end of the string if there is room or if the array dimensions are not specified.

The following definitions show character array initializations:

```

static char name1[ ] = { 'J', 'a', 'n' };
static char name2[ ] = { "Jan" };
static char name3[4] = "Jan";

```

These definitions create the following elements:

Element	Value	Element	Value	Element	Value
name1[0]	J	name2[0]	J	name3[0]	J
name1[1]	a	name2[1]	a	name3[1]	a
name1[2]	n	name2[2]	n	name3[2]	n
		name2[3]	\0	name3[3]	\0

Note that the following definition would result in the null character being lost:

```
static char name3[3]="Jan";
```

C++ In C++, when initializing an array of characters with a string, the number of characters in the string — including the terminating `'\0'` — must not exceed the number of elements in the array.

You can initialize a multidimensional array by:

- Listing the values of all elements you want to initialize, in the order that the compiler assigns the values. The compiler assigns values by increasing the subscript of the last dimension fastest. This form of a multidimensional array initialization looks like a one-dimensional array initialization. The following definition completely initializes the array `month_days`:

```

static month_days[2][12] =
{
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
    31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};

```

- Using braces to group the values of the elements you want initialized. You can put braces around each element, or around any nesting level of elements. The following definition contains two elements in the first dimension (you can consider these elements as rows). The initialization contains braces around each of these two elements:

```

static int month_days[2][12] =
{
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
    { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};

```

C/C++ Data Mapping

- Using nested braces to initialize dimensions and elements in a dimension selectively.

The following definition explicitly initializes six elements in a 12-element array:

```
static int matrix[3][4] =
{
    {1, 2},
    {3, 4},
    {5, 6}
};
```

The initial values of `matrix` are:

Element	Value	Element	Value
<code>matrix[0][0]</code>	1	<code>matrix[1][2]</code>	0
<code>matrix[0][1]</code>	2	<code>matrix[1][3]</code>	0
<code>matrix[0][2]</code>	0	<code>matrix[2][0]</code>	5
<code>matrix[0][3]</code>	0	<code>matrix[2][1]</code>	6
<code>matrix[1][0]</code>	3	<code>matrix[2][2]</code>	0
<code>matrix[1][1]</code>	4	<code>matrix[2][3]</code>	0

You cannot have more initializers than the number of elements in the array.

RELATED REFERENCES

- “Initializers” on page 79
- “Arrays” on page 86
- “extern Storage Class Specifier” on page 37
- “static Storage Class Specifier” on page 42

Example Programs Using Arrays

The following program defines a floating-point array called `prices`.

The first `for` statement prints the values of the elements of `prices`. The second `for` statement adds five percent to the value of each element of `prices`, and assigns the result to `total`, and prints the value of `total`.

CCNRAAO

```
/**
** Example of one-dimensional arrays
**/

#include <stdio.h>
#define ARR_SIZE 5

int main(void)
{
    static float const prices[ARR_SIZE] = { 1.41, 1.50, 3.75, 5.00, .86 };
    auto float total;
    int i;

    for (i = 0; i < ARR_SIZE; i++)
    {
        printf("price = %.2f\n", prices[i]);
    }

    printf("\n");

    for (i = 0; i < ARR_SIZE; i++)
```

```

    {
        total = prices[i] * 1.05;
        printf("total = $%.2f\n", total);
    }

    return(0);
}

```

This program produces the following output:

```

price = $1.41
price = $1.50
price = $3.75
price = $5.00
price = $0.86

```

```

total = $1.48
total = $1.57
total = $3.94
total = $5.25
total = $0.90

```

The following program defines the multidimensional array `salary_tbl`. A **for** loop prints the values of `salary_tbl`.

```

/**
 ** Example of a multidimensional array
 **/

#include <stdio.h>
#define ROW_SIZE    3
#define COLUMN_SIZE 5

int main(void)
{
    static int
    salary_tbl[ROW_SIZE][COLUMN_SIZE] =
    {
        { 500, 550, 600, 650, 700 },
        { 600, 670, 740, 810, 880 },
        { 740, 840, 940, 1040, 1140 }
    };
    int grade, step;

    for (grade = 0; grade < ROW_SIZE; ++grade)
        for (step = 0; step < COLUMN_SIZE; ++step)
        {
            printf("salary_tbl[%d]
[%d] = %d\n", grade, step,
                salary_tbl[grade][step]);
        }

    return(0);
}

```

This program produces the following output:

```

salary_tbl[0][0] = 500
salary_tbl[0][1] = 550
salary_tbl[0][2] = 600
salary_tbl[0][3] = 650
salary_tbl[0][4] = 700
salary_tbl[1][0] = 600
salary_tbl[1][1] = 670
salary_tbl[1][2] = 740
salary_tbl[1][3] = 810
salary_tbl[1][4] = 880
salary_tbl[2][0] = 740

```

C/C++ Data Mapping

```
salary_tbl[2][1] = 840
salary_tbl[2][2] = 940
salary_tbl[2][3] = 1040
salary_tbl[2][4] = 1140
```

RELATED REFERENCES

- “Arrays” on page 86
- “Pointers” on page 81
- “Array Subscript [] (Array Element Specification)” on page 106
- “String Literals” on page 29
- “Chapter 4. Declarators” on page 73
- “Initializers” on page 79
- “Chapter 6. Implicit Type Conversions” on page 143

Function Specifiers

➤ **C++** The function specifiers **inline** and **virtual** are used only in C++ function declarations.

The function specifier **inline** is used to make a suggestion to the compiler to incorporate the code of a function into the code at the point of the call.

The function specifier **virtual** can only be used in nonstatic member function declarations.

RELATED REFERENCES

- “Function Declarations” on page 154
- “Inline Functions” on page 174
- “Virtual Functions” on page 333

References

A *reference* is an alias or an alternative name for an object. All operations applied to a reference act on the object the reference refers to. The address of a reference is the address of the aliased object.

A reference type is defined by placing the & after the type specifier. You must initialize all references except function parameters when they are defined.

➤ **C++** Because arguments of a function are passed by value, a function call does not modify the actual values of the arguments. If a function needs to modify the actual value of an argument, the argument must be passed by *reference* (as opposed to being passed by *value*). Passing arguments by reference can be done using either references or pointers. In C++, this is accomplished transparently.

➤ **C++** Unlike C, C++ does not force you to use pointers if you want to pass arguments by reference. For example:

```
int f(int&);
int main()
{
    extern int i;
    f(i);
}
```

You cannot tell from the function call `f(i)` that the argument is being passed by reference.

References to `NULL` are not allowed.

RELATED REFERENCES

- “Objects” on page 34
- “Function Calls ()” on page 104
- “Pointers” on page 81

Initializing References

The object that you use to initialize a reference must be of the same type as the reference, or it must be of a type that is convertible to the reference type. If you initialize a reference to a constant using an object that requires conversion, a temporary object is created. In the following example, a temporary object of type **float** is created:

```
int i;
const float& f = i; // reference to a constant float
```

When you initialize a reference with an object, you *bind* that reference to that object.

Attempting to initialize a nonconstant reference with an object that requires a conversion is an error.

Once a reference has been initialized, it cannot be modified to refer to another object. For example:

```
int num1 = 10;
int num2 = 20;

int &RefOne = num1;           // valid
int &RefOne = num2;           // error, two definitions of RefOne
RefOne = num2;                // assign num2 to num1
int &RefTwo;                  // error, uninitialized reference
int &RefTwo = num2;           // valid
```

Note that the initialization of a reference is not the same as an assignment to a reference. Initialization operates on the actual reference by initializing the reference with the object it is an alias for. Assignment operates through the reference on the object referred to.

A reference can be declared without an initializer:

- When it is used in an parameter declaration
- In the declaration of a return type for a function call
- In the declaration of class member within its class declaration
- When the **extern** specifier is explicitly used

You cannot have references to any of the following:

- Other references
- Bit fields
- Arrays of references
- Pointers to references

Direct Binding

Suppose a reference `r` of type `T` is initialized by an expression `e` of type `U`.

References

The reference *r* is *bound directly* to *e* if the following statements are true:

- Expression *e* is an lvalue
- *T* is the same type as *U*, or *T* is a base class of *U*
- *T* has the same, or more, **const** or **volatile** qualifiers than *U*

The reference *r* is also bound directly to *e* if *e* can be implicitly converted to a type such that the previous list of statements is true.

RELATED REFERENCES

- “References” on page 92
- “Declaring Class Types” on page 283
- “Passing Arguments by Reference” on page 167
- “Pointers” on page 81
- “extern Storage Class Specifier” on page 37
- “volatile and const Qualifiers” on page 69
- “Chapter 4. Declarators” on page 73
- “Initializers” on page 79
- “Temporary Objects” on page 357

Chapter 5. Expressions and Operators

Expressions are sequences of operators, operands, and punctuators that specify a computation. The evaluation of expressions is based on the operators that the expressions contain and the context in which they are used.

An expression can result in an lvalue, rvalue, or no value, and can produce side effects in each case.

C++ C++ operators can be defined to behave differently when applied to operands of class type. This is called operator *overloading*. This chapter describes the behavior of operators that are not overloaded.

RELATED REFERENCES

- “Lvalues and Rvalues” on page 99
- “Overloading Operators” on page 271

Operator Precedence and Associativity

Two operator characteristics determine how operands group with operators: *precedence* and *associativity*. Precedence is the priority for grouping different types of operators with their operands. Associativity is the left-to-right or right-to-left order for grouping operands to operators that have the same precedence. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first.

For example, in the following statements, the value of 5 is assigned to both a and b because of the right-to-left associativity of the = operator. The value of c is assigned to b first, and then the value of b is assigned to a.

```
b = 9;  
c = 5;  
a = b = c;
```

Because the order of subexpression evaluation is not specified, you can explicitly force the grouping of operands with operators by using parentheses.

In the expression

```
a + b * c / d
```

the * and / operations are performed before + because of precedence. b is multiplied by c before it is divided by d because of associativity.

The following table lists the C and C++ language operators in order of precedence and shows the direction of associativity for each operator.













The C++ scope resolution operator (::) has the highest precedence. The comma operator has the lowest precedence. Operators that have the same rank have the same precedence.

Table 3. Precedence and associativity of C and C++ operators

Rank	Right Associative?	Operator Function	Usage
1	yes	C++ global scope resolution	:: <i>name_or_qualified name</i>

Operator Precedence and Associativity

Table 3. Precedence and associativity of C and C++ operators (continued)

Rank	Right Associative?	Operator Function	Usage
1		 class or namespace scope resolution	<i>class_or_namespace :: member</i>
2		member selection	<i>object . member</i>
2		member selection	<i>pointer -> member</i>
2		subscripting	<i>pointer [expr]</i>
2		function call	<i>expr (expr_list)</i>
2		value construction	<i>type (expr_list)</i>
2		postfix increment	<i>lvalue ++</i>
2		postfix decrement	<i>lvalue --</i>
2	yes	 type identification	typeid (<i>type</i>)
2	yes	 type identification at run time	typeid (<i>expr</i>)
2	yes	 conversion checked at compile time	static_cast < <i>type</i> > (<i>expr</i>)
2	yes	 conversion checked at run time	dynamic_cast < <i>type</i> > (<i>expr</i>)
2	yes	 unchecked conversion	reinterpret_cast < <i>type</i> > (<i>expr</i>)
2	yes	 const conversion	const_cast < <i>type</i> > (<i>expr</i>)
3	yes	size of object in bytes	sizeof (<i>expr</i>)
3	yes	size of type in bytes	sizeof <i>type</i>
3	yes	prefix increment	++ <i>lvalue</i>
3	yes	prefix decrement	-- <i>lvalue</i>
3	yes	complement	~ <i>expr</i>
3	yes	not	! <i>expr</i>
3	yes	unary minus	- <i>expr</i>
3	yes	unary plus	+ <i>expr</i>
3	yes	address of	& <i>lvalue</i>
3	yes	indirection or dereference	* <i>expr</i>
3	yes	 create (allocate memory)	new <i>type</i>
3	yes	 create (allocate and initialize memory)	new <i>type</i> (<i>expr_list</i>) <i>type</i>
3	yes	 create (placement)	new <i>type</i> (<i>expr_list</i>) <i>type</i> (<i>expr_list</i>)
3	yes	 destroy (deallocate memory)	delete <i>pointer</i>
3	yes	 destroy array	delete [] <i>pointer</i>
3	yes	type conversion (cast)	(<i>type</i>) <i>expr</i>
4		member selection	<i>object . * ptr_to_member</i>

Operator Precedence and Associativity

Table 3. Precedence and associativity of C and C++ operators (continued)

Rank	Right Associative?	Operator Function	Usage
4		member selection	<i>object ->* ptr_to_member</i>
5		multiplication	<i>expr * expr</i>
5		division	<i>expr / expr</i>
5		modulo (remainder)	<i>expr % expr</i>
6		binary addition	<i>expr + expr</i>
6		binary subtraction	<i>expr - expr</i>
7		bitwise shift left	<i>expr << expr</i>
7		bitwise shift right	<i>expr >> expr</i>
8		less than	<i>expr < expr</i>
8		less than or equal to	<i>expr <= expr</i>
8		greater than	<i>expr > expr</i>
8		greater than or equal to	<i>expr >= expr</i>
9		equal	<i>expr == expr</i>
9		not equal	<i>expr != expr</i>
10		bitwise AND	<i>expr & expr</i>
11		bitwise exclusive OR	<i>expr ^ expr</i>
12		bitwise inclusive OR	<i>expr expr</i>
13		logical AND	<i>expr && expr</i>
14		logical inclusive OR	<i>expr expr</i>
16	yes	simple assignment	<i>lvalue = expr</i>
16	yes	multiply and assign	<i>lvalue *= expr</i>
16	yes	divide and assign	<i>lvalue /= expr</i>
16	yes	modulo and assign	<i>lvalue %= expr</i>
16	yes	add and assign	<i>lvalue += expr</i>
16	yes	subtract and assign	<i>lvalue -= expr</i>
16	yes	shift left and assign	<i>lvalue <<= expr</i>
16	yes	shift right and assign	<i>lvalue >>= expr</i>
16	yes	bitwise AND and assign	<i>lvalue &= expr</i>
16	yes	bitwise exclusive OR and assign	<i>lvalue ^= expr</i>
16	yes	bitwise inclusive OR and assign	<i>lvalue = expr</i>
16		conditional expression	<i>expr ? expr : expr</i>
17	yes	 throw expression	throw <i>expr</i>
18		comma (sequencing)	<i>expr , expr</i>

The order of evaluation for function call arguments or for the operands of binary operators is not specified. Avoid writing such ambiguous expressions as:

```
z = (x * ++y) / func1(y);
func2(++i, x[i]);
```

In the example above, `++y` and `func1(y)` might not be evaluated in the same order by all C language implementations. If `y` had the value of 1 before the first

Operator Precedence and Associativity

statement, it is not known whether or not the value of 1 or 2 is passed to `func1()`. In the second statement, if `i` had the value of 1, it is not known whether the first or second array element of `x[]` is passed as the second argument to `func2()`.

The order of grouping operands with operators in an expression containing more than one instance of an operator with both associative and commutative properties is not specified. The operators that have the same associative and commutative properties are: `*`, `+`, `&`, `|`, and `^` (or `~`). The grouping of operands can be forced by grouping the expression in parentheses.

Examples of Expressions and Precedence

The parentheses in the following expressions explicitly show how the compiler groups operands and operators.

```
total = (4 + (5 * 3));  
total = (((8 * 5) / 10) / 3);  
total = (10 + (5/3));
```

If parentheses did not appear in these expressions, the operands and operators would be grouped in the same manner as indicated by the parentheses. For example, the following expressions produce the same output.

```
total = (4+(5*3));  
total = 4+5*3;
```

Because the order of grouping operands with operators that are both associative and commutative is not specified, the compiler can group the operands and operators in the expression:

```
total = price + prov_tax +  
city_tax;
```

in the following ways (as indicated by parentheses):

```
total = (price + (prov_tax + city_tax));  
total = ((price + prov_tax) + city_tax);  
total = ((price + city_tax) + prov_tax);
```

The grouping of operands and operators does not affect the result unless one ordering causes an overflow and another does not. For example, if `price = 32767`, `prov_tax = -42`, and `city_tax = 32767`, and all three of these variables have been declared as integers, the third statement `total = ((price + city_tax) + prov_tax)` will cause an integer overflow and the rest will not.

Because intermediate values are rounded, different groupings of floating-point operators may give different results.

In certain expressions, the grouping of operands and operators can affect the result. For example, in the following expression, each function call might be modifying the same global variables.

```
a = b() + c() + d();
```

This expression can give different results depending on the order in which the functions are called.

If the expression contains operators that are both associative and commutative and the order of grouping operands with operators can affect the result of the expression, separate the expression into several expressions. For example, the following expressions could replace the previous expression if the called functions do not produce any side effects that affect the variable `a`.

```
a = b();
a += c();
a += d();
```

Lvalues and Rvalues

Every expression is either an *lvalue* or an *rvalue*.

An *lvalue* is an expression or function that represents an object that can be examined or changed. A *modifiable lvalue* is an expression representing an object that can be changed. It is typically the left operand in an assignment expression. For example, arrays and **const** objects are not modifiable lvalues, but **static int** objects are.

An *rvalue* is an expression that cannot have a value assigned to it. A function that does not return a reference yields an rvalue. Rvalues always have complete types or the void type.

Certain operators require lvalues for some of their operands. For example, all assignment operators evaluate their right operand and assign that value to their left operand. The left operand must evaluate to a reference to an object. The address operator (&) requires an lvalue as an operand while the increment (++) and the decrement (--) operators require a modifiable lvalue as an operand.

The following are examples of lvalues:

Expression	Lvalue
<code>x = 42</code>	<code>x</code>
<code>*ptr = newvalue</code>	<code>*ptr</code>
<code>a++</code>	<code>a</code>
<code>int& f()</code>	The function call to <code>f()</code>

RELATED REFERENCES

- “Arrays” on page 86
- “volatile and const Qualifiers” on page 69
- “static Storage Class Specifier” on page 42

Type-Based Aliasing

z/OS The compiler follows the type-based aliasing rule in the ISO C standard when the `ANSIALIAS` option is in effect. This rule, also known as the ANSI aliasing rule, states that a pointer can only be dereferenced to an object of the same type.¹ The common coding practice of casting a pointer to a different type and then dereferencing it violates this rule. Note that char pointers are an exception to this rule. Refer to the description of the `ANSIALIAS` option in *z/OS C/C++ User's Guide* for additional information.

The compiler uses the type-based aliasing information to perform optimizations to the generated code. Contravening the type-based aliasing rule can lead to unexpected behavior, as demonstrated in the following example:

```
int *p;
double d = 0.0;

int *faa(double *g); /* cast operator inside the function */

void foo(double f) {
    p = faa(&f);      /* turning &f into a int ptr */
    f += 1.0;          /* compiler may purge this statement */
    printf("f=%x\n", *p);
}

int *faa(double *g) { return (int*) g; } /* questionable cast; */
/* the function can be
/* in another compile
unit */

int main() {
    foo(0.0);
}
```

In the above `printf` statement, `*p` can not be dereferenced to a `double` under the ANSI aliasing rule. The compiler determines that the result of `f += 1.0;` is never

1. The C standard states:

An object shall have its stored value accessed only by an lvalue that has one of the following types:

- the declared type of the object,
- a qualified version of the declared type of the object,
- a type that is the signed or unsigned type corresponding to the declared type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type

The C++ standard states that if a program attempts to access the stored value of an object through an lvalue of other than one of the following types, the behavior is undefined:

- the dynamic type of the object,
- a cv-qualified version of the dynamic type of the object,
- a type that is the signed or unsigned type corresponding to the dynamic type of the object,
- a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union),
- a type that is a (possible cv-qualified) base class type of the dynamic type of the object,
- a char or unsigned char type.

used subsequently. Thus, the optimizer may purge the statement from the generated code. If you compile the above example with the `OPTIMIZE` and `ANSIALIAS` options, the `printf` statement may output 0 (zero).

RELATED REFERENCES

- “`reinterpret_cast` Operator” on page 109

Integer Constant Expressions

An *integer compile-time constant* is a value that is determined during compilation and cannot be changed at run time. An *integer compile-time constant expression* is an expression that is composed of constants and evaluated to a constant.

An integer constant expression is an expression that is composed of only the following:

- literals
- enumerators
- `const` variables
- static data members of integral or enumeration types
- casts to integral types
- **`sizeof`** expressions

You must use an integer constant expression in the following situations:

- In the subscript declarator as the description of an array bound
- After the keyword **`case`** in a **`switch`** statement
- In an enumerator, as the numeric value of an enum constant
- In a bit-field width specifier
- In the preprocessor **`#if`** statement (Enumeration constants, address constants, and **`sizeof`** cannot be specified in the preprocessor **`#if`** statement)

RELATED REFERENCES

- “Literals” on page 23
- “Enumerations” on page 65
- “volatile and `const` Qualifiers” on page 69
- “Static Members” on page 303
- “Cast Expressions” on page 135
- “`sizeof` (Size of an Object)” on page 117

Primary Expressions

Primary expressions are literals, the C++ **`this`** pointer, parenthesized expressions, names, and names qualified by the scope resolution operator (`::`).

RELATED REFERENCES

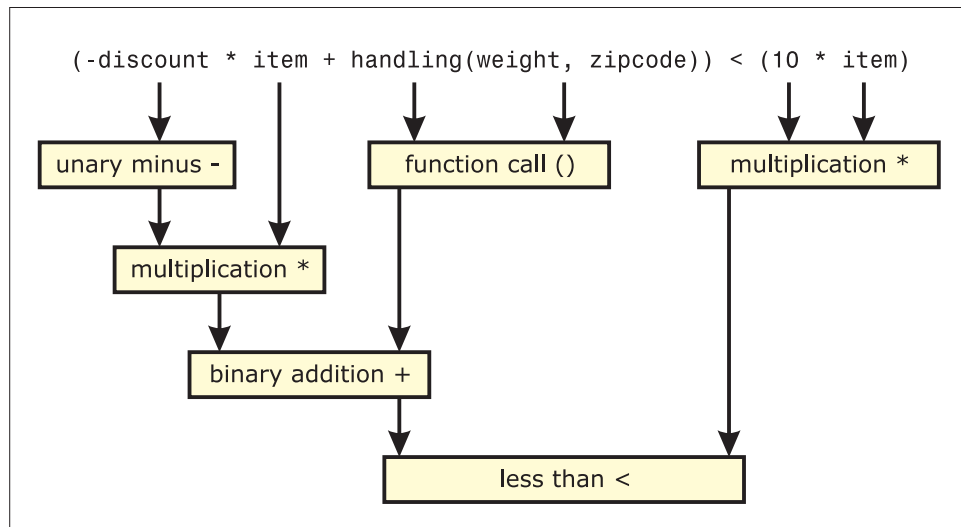
- “Literals” on page 23
- “The `this` Pointer” on page 300

Parenthesized Expressions ()

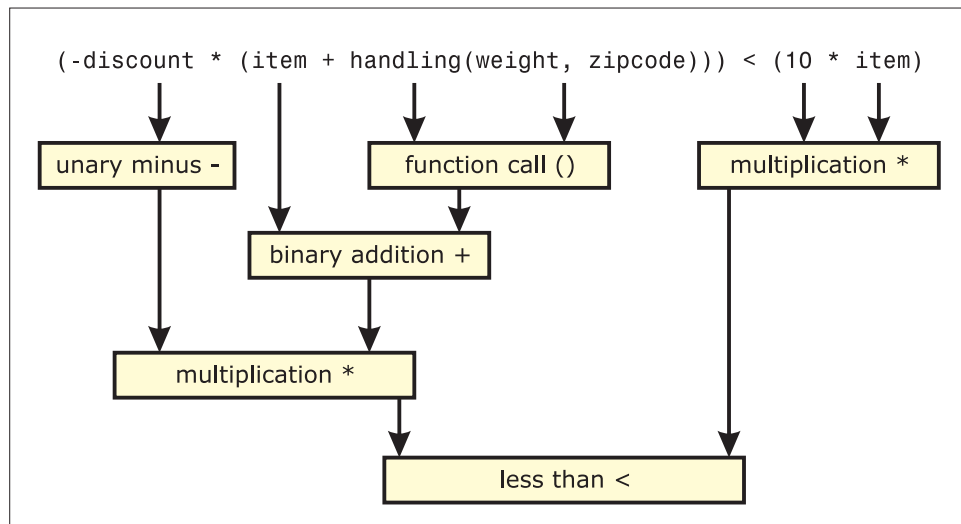
Use parentheses to explicitly force the order of expression evaluation. The following expression does not contain any parentheses used for grouping operands and operators. The parentheses surrounding `weight`, `zipcode` are used to form a function call. Note how the compiler groups the operands and operators in the

lvalue

expression according to the rules for operator precedence and associativity:



The following expression is similar to the previous expression, but it contains parentheses that change how the operands and operators are grouped:



In an expression that contains both associative and commutative operators, you can use parentheses to specify the grouping of operands with operators. The parentheses in the following expression guarantee the order of grouping operands with the operators:

```
x = f + (g + h);
```

C++ Scope Resolution Operator ::

C++ The `::` (scope resolution) operator is used to qualify hidden names so that you can still use them. You can use the unary scope operator if a namespace scope or global scope name is hidden by an explicit declaration of the same name in a block or class. For example:

```
int count = 0;

int main(void) {
```



```

int count = 0;
::count = 1; // set global count to 1
count = 2;   // set local count to 2
return 0;
}

```

The declaration of `count` declared in the `main()` function hides the integer named `count` declared in global namespace scope. The statement `::count = 1` accesses the variable named `count` declared in global namespace scope.

You can also use the class scope operator to qualify class names or class member names. If a class member name is hidden, you can use it by qualifying it with its class name and the class scope operator.

In the following example, the declaration of the variable `X` hides the class type `X`, but you can still use the static class member `count` by qualifying it with the class type `X` and the scope resolution operator.

```

#include <iostream>
using namespace std;

class X
{
public:
    static int count;
};
int X::count = 10;           // define static data member

int main ()
{
    int X = 0;               // hides class type X
    cout << X::count << endl; // use static member of class X
}

```


RELATED REFERENCES

- “Scope of Class Names” on page 287
- “Chapter 10. Namespaces” on page 261

Postfix Expressions

Postfix operators are operators that appear after their operands. A *postfix expression* is a primary expression, or a primary expression that contains a postfix operator. The following summarizes the available postfix operators:

Table 4. Precedence and associativity of postfix operators

Rank	Right Associative?	Operator Function	Usage
2		member selection	<i>object . member</i>
2		member selection	<i>pointer -> member</i>
2		subscripting	<i>pointer [expr]</i>
2		function call	<i>expr (expr_list)</i>
2		value construction	<i>type (expr_list)</i>
2		postfix increment	<i>lvalue ++</i>
2		postfix decrement	<i>lvalue --</i>
2	yes	 type identification	typeid (<i>type</i>)

Ivalue

Table 4. Precedence and associativity of postfix operators (continued)

Rank	Right Associative?	Operator Function	Usage
2	yes	<code>> C++</code> type identification at run time	typeid (<i>expr</i>)
2	yes	<code>> C++</code> conversion checked at compile time	static_cast < <i>type</i> > (<i>expr</i>)
2	yes	<code>> C++</code> conversion checked at run time	dynamic_cast < <i>type</i> > (<i>expr</i>)
2	yes	<code>> C++</code> unchecked conversion	reinterpret_cast < <i>type</i> > (<i>expr</i>)
2	yes	<code>> C++</code> const conversion	const_cast < <i>type</i> > (<i>expr</i>)

Function Calls ()

A *function call* is an expression containing a simple type name and a parenthesized argument list. The argument list can contain any number of expressions separated by commas. It can also be empty.

For example:

```
stub()
overdue(account, date, amount)
notify(name, date + 5)
report(error, time, date, ++num)
```

There are two kinds of function calls: ordinary function calls and C++ member function calls. Any function may call itself except for the function **main**.

Type of a Function Call

The type of a function call expression is the return type of the function. This type can either be a complete type, a reference type, or the type **void**. A function call is an lvalue if and only if the type of the function is a reference.

Arguments and Parameters

A *function argument* is an expression that you use within the parenthesis of a function call. A *function parameter* is an object or reference declared within the parenthesis of a function declaration or definition. When you call a function, the arguments are evaluated, and each parameter is initialized with the value of the corresponding argument. The semantics of argument passing are identical to those of assignment.

A function can change the values of its non-const parameters, but these changes have no effect on the argument unless the parameter is a reference type.

If you want a function to change the value of a variable, pass a pointer or a reference to the variable you want changed. When a pointer is passed as a parameter, the pointer is copied; the object pointed to is not copied.

Linkage and Function Calls

► C In C only, if a function definition has external linkage and a return type of **int**, calls to the function can be made before it is explicitly declared because an implicit declaration of `extern int func();` is assumed. This is *not* true for C++.

Type Conversions of Arguments

Arguments that are arrays and functions are converted to pointers before being passed as function arguments.

Arguments passed to nonprototyped C functions undergo conversions: type **short** or **char** parameters are converted to **int**, and **float** parameters to **double**. Use a cast expression for other conversions.

The compiler compares the data types provided by the calling function with the data types that the called function expects and performs necessary type conversions. For example, when function `func` is called, argument `f` is converted to a **double**, and argument `c` is converted to an **int**:

```
char * func (double d, int i);
    /* ... */
int main(void)
{
    float f;
    char c;
    func(f, c) /* f is a double, c is an int */
    return 0;
}
```

Evaluation Order of Arguments

The order in which arguments are evaluated is not specified. Avoid such calls as:
`method(sample1, batch.process--, batch.process);`

In this example, `batch.process--` might be evaluated last, causing the last two arguments to be passed with the same value.

Example of Function Calls

In the following example, `main` passes `func` two values: 5 and 7. The function `func` receives copies of these values and accesses them by the identifiers: `a` and `b`. The function `func` changes the value of `a`. When control passes back to `main`, the actual values of `x` and `y` are not changed. The called function `func` only receives copies of the values of `x` and `y`, not the variables themselves.

```
/**
 ** This example illustrates function calls
 **/

#include <stdio.h>

void func (int a, int b)
{
    a += b;
    printf("In func, a = %d    b = %d\n", a, b);
}

int main(void)
{
    int x = 5, y = 7;
```

Ivalue

```
func(x, y);  
printf("In main, x = %d    y = %d\n", x, y);  
return 0;  
}
```

This program produces the following output:

```
In func, a = 12    b = 7  
In main, x = 5     y = 7
```

F

RELATED REFERENCES

- “Chapter 7. Functions” on page 153
- “Cast Expressions” on page 135
- “Pointers” on page 81
- “Program Linkage” on page 5

Array Subscript [] (Array Element Specification)

A postfix expression followed by an expression in [] (square brackets) specifies an element of an array. The expression within the square brackets is referred to as a *subscript*.

The expression `a[b]` is equivalent (by definition) to the expression `*((a) + (b))` (and because addition is associative, it is also equivalent to `b[a]`). Between expressions `a` and `b`, one must be a pointer to a type `T`, and the other must have integral or enumeration type. The result of an array subscript is an lvalue. The following example demonstrates this:

```
#include <stdio.h>  
  
int main(void) {  
    int a[3] = { 10, 20, 30 };  
    printf("a[0] = %d\n", a[0]);  
    printf("a[1] = %d\n", 1[a]);  
    printf("a[2] = %d\n", *(2 + a));  
    return 0;  
}
```

The following is the output of the above example:

```
a[0] = 10  
a[1] = 20  
a[2] = 30
```

C++ The above restrictions on the types of expressions required by the subscript operator, as well as the relationship between the subscript operator and pointer arithmetic, do not apply if you overload **operator[]** of a class.

The first element of each array has the subscript 0. The expression `contract[35]` refers to the 36th element in the array `contract`.

In a multidimensional array, you can reference each element (in the order of increasing storage locations) by incrementing the right-most subscript most frequently.

For example, the following statement gives the value 100 to each element in the array `code[4][3][6]`:

```

for (first = 0; first < 4; ++first)
{
    for (second = 0; second < 3; ++second)
    {
        for (third = 0; third < 6; ++third)
        {
            code[first][second][third] =
            100;
        }
    }
}

```

RELATED REFERENCES

- “Pointers” on page 81
- “Integer Variables” on page 49
- “Lvalues and Rvalues” on page 99
- “Arrays” on page 86
- “Overloading Subscripting” on page 277
- “Pointer Arithmetic” on page 84

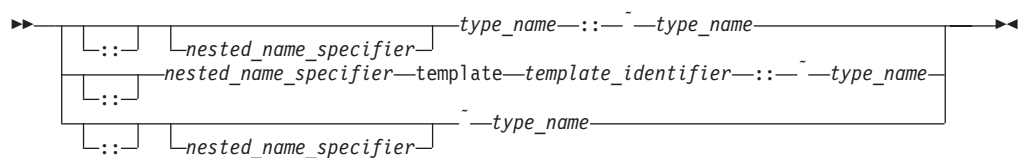
Dot Operator .

The . (dot) operator is used to access class, structure, or union members. The member is specified by a postfix expression, followed by a . (dot) operator, followed by a possibly qualified identifier or a pseudo-destructor name. The postfix expression must be an object of type **class**, **struct** or **union**. The name must be a member of that object.

The value of the expression is the value of the selected member. If the postfix expression and the name are lvalues, the expression value is also an lvalue.

Pseudo-destructors

C++ A *pseudo-destructor* is a destructor of a nonclass type named *type_name* in the following syntax diagram :

**RELATED REFERENCES**

- “Chapter 13. Class Members and Friends” on page 293
- “Unions” on page 59
- “Structures” on page 51
- “Scope of Class Names” on page 287

Arrow Operator ->

The -> (arrow) operator is used to access class, structure or union members using a pointer. A postfix expression, followed by an -> (arrow) operator, followed by a possibly qualified identifier or a pseudo-destructor name, designates a member of the object to which the pointer points. (A *pseudo-destructor* is a destructor of a nonclass type.) The postfix expression must be a pointer to an object of type **class**, **struct** or **union**. The name must be a member of that object.

Ivalue

The value of the expression is the value of the selected member. If the name is an lvalue, the expression value is also an lvalue.

RELATED REFERENCES

- “Pointers” on page 81
- “Chapter 13. Class Members and Friends” on page 293
- “Unions” on page 59
- “Structures” on page 51
- “Dot Operator .” on page 107

static_cast Operator

 The *static_cast operator* converts a given expression to a specified type.

Syntax — static_cast

►► static_cast ◀◀ *Type* ◀◀ (◀◀ *expression* ◀◀) ◀◀

The following is an example of the **static_cast** operator.

```
#include <iostream>
using namespace std;

int main() {
    int j = 41;
    int v = 4;
    float m = j/v;
    float d = static_cast<float>(j)/v;
    cout << "m = " << m << endl;
    cout << "d = " << d << endl;
}
```

The following is the output of the above example:

```
m = 10
d = 10.25
```

In this example, `m = j/v;` produces an answer of type `int` because both `j` and `v` are integers. Conversely, `d = static_cast<float>(j)/v;` produces an answer of type `float`. The **static_cast** operator converts variable `j` to a type **float**. This allows the compiler to generate a division with an answer of type **float**. All **static_cast** operators resolve at compile time and do not remove any **const** or **volatile** modifiers.

Applying the **static_cast** operator to a null pointer will convert it to a null pointer value of the target type.

You can explicitly convert a pointer of a type `A` to a pointer of a type `B` if `A` is a base class of `B`. If `A` is not a base class of `B`, a compiler error will result.

You may cast an lvalue of a type `A` to a type `B` if the following are true:

- `A` is a base class of `B`
- You are able to convert a pointer of type `A` to a pointer of type `B`
- The type `B` has the same or greater **const** or **volatile** qualifiers than type `A`
- `A` is not a virtual base class of `B`

The result is an lvalue of type `B`.

A pointer to member type can be explicitly converted into a different pointer to member type if both types are pointers to members of the same class. This form of explicit conversion may also take place if the pointer to member types are from separate classes, however one of the class types must be derived from the other.

reinterpret_cast Operator

► C++ A *reinterpret_cast* operator handles conversions between unrelated types.

Syntax — reinterpret_cast

► reinterpret_cast<Type> (expression) ◀

The *reinterpret_cast* operator produces a value of a new type that has the same bit pattern as its argument. You cannot cast away a *const* or *volatile* qualification. You can explicitly perform the following conversions:

- A pointer to any integral type large enough to hold it
- A value of integral or enumeration type to a pointer
- A pointer to a function to a pointer to a function of a different type
- A pointer to an object to a pointer to an object of a different type
- A pointer to a member to a pointer to a member of a different class or type, if the types of the members are both function types or object types

A null pointer value is converted to the null pointer value of the destination type.

Given an lvalue expression of type *T* and an object *x*, the following two conversions are synonymous:

- `reinterpret_cast<T*>(x)`
- `*reinterpret_cast<T*>(&x)`

ISO C++ also supports C-style casts. The two styles of explicit casts have different syntax but the same semantics, and either way of reinterpreting one type of pointer as an incompatible type of pointer is usually invalid. The *reinterpret_cast* operator, as well as the other named cast operators, is more easily spotted than C-style casts, and highlights the paradox of a strongly typed language that allows explicit casts.

The C++ compiler detects and quietly fixes most but not all violations. It is important to remember that even though a program compiles, its source code may not be completely correct. On some platforms, performance optimizations are predicated on strict adherence to ISO aliasing rules. Although the C++ compiler tries to help with type-based aliasing violations, it cannot detect all possible cases.

The following example violates the aliasing rule, but will execute as expected when compiled unoptimized in C++ or in K&R C or with `NOANSIALIAS`. It will also successfully compile optimized in C++ with `ANSIALIAS`, but will not necessarily execute as expected. The offending line 7 causes an old or uninitialized value for *x* to be printed.

```
1 extern int y = 7.;
2
3 void main() {
4     float x;
5     int i;
```

lvalue

```
6     x = y;
7     i = *(int *) &x;
8     printf("i=%d. x=%f.\n", i, x);
9 }
```

The next code example contains an incorrect cast that the compiler cannot even detect because the cast is across two different files.


```
1  /* separately compiled file 1 */
2      extern float f;
3      extern int * int_pointer_to_f = (int *) &f; /* suspicious cast */
4
5  /* separately compiled file 2 */
6      extern float f;
7      extern int * int_pointer_to_f;
8      f = 1.0;
9      int i = *int_pointer_to_f;           /* no suspicious cast but wrong */
```

In line 8, there is no way for the compiler to know that `f = 1.0` is storing into the same object that `int i = *int_pointer_to_f` is loading from.

RELATED REFERENCES

- “Type-Based Aliasing” on page 100

const_cast Operator

 A *const_cast operator* is used to add or remove a `const` or `volatile` modifier to or from a type.

Syntax — const_cast

►►—const_cast—<—Type—>—(—expression—)—————►►

Type and the type of *expression* may only differ with respect to their **const** and **volatile** qualifiers. Their cast is resolved at compile time. A single **const_cast** expression may add or remove any number of **const** or **volatile** modifiers.

The result of a **const_cast** expression is an rvalue unless *Type* is a reference type. In this case, the result is an lvalue.

Types can not be defined within **const_cast**.

The following demonstrates the use of the **const_cast** operator:

```
#include <iostream>
using namespace std;

void f(int* p) {
    cout << *p << endl;
}

int main(void) {
    const int a = 10;
    const int* b = &a;

    // Function f() expects int*, not const int*
    // f(b);
    int* c = const_cast<int*>(b);
    f(c);
}
```



```

// Lvalue is const
// *b = 20;

// Undefined behavior
// *c = 30;

int a1 = 40;
const int* b1 = &a1;
int* c1 = const_cast<int>(b1);

// Integer a1, the object referred to by c1, has
// not been declared const
*c1 = 50;

return 0;
}

```

The compiler will not allow the function call `f(b)`. Function `f()` expects a pointer to an **int**, not a **const int**. The statement `int* c = const_cast<int>(b)` returns a pointer `c` that refers to `a` without the **const** qualification of `a`. This process of using **const_cast** to remove the **const** qualification of an object is called *casting away constness*. Consequently the compiler will allow the function call `f(c)`.

The compiler would not allow the assignment `*b = 20` because `b` points to an object of type **const int**. The compiler will allow the `*c = 30`, but the behavior of this statement is undefined. If you cast away the constness of an object that has been explicitly declared as **const**, and attempt to modify it, the results are undefined.

However, if you cast away the constness of an object that has not been explicitly declared as **const**, you can modify it safely. In the above example, the object referred to by `b1` has not been declared **const**, but you cannot modify this object through `b1`. You may cast away the constness of `b1` and modify the value to which it refers.

dynamic_cast Operator

► C++ The *dynamic_cast* operator performs type conversions at run time.

The expression `dynamic_cast<T>(v)` converts the expression `v` to type `T`. Type `T` must be a pointer or reference to a complete class type or a pointer to void. If `T` is a pointer and the **dynamic_cast** operator fails, the operator returns a null pointer of type `T`. If `T` is a reference and the **dynamic_cast** operator fails, the operator throws the exception `std::bad_cast`. You can find this class in the standard library header `<typeinfo>`.

If `T` is a void pointer, then **dynamic_cast** will return the starting address of the object pointed to by `v`. The following example demonstrates this:

```

#include <iostream>
using namespace std;

struct A {
    virtual ~A() { };
};

struct B : A { };

int main() {
    B bobj;
    A* ap = &bobj;
}

```

Ivalue

```
void * vp = dynamic_cast<void>(ap);
cout << "Address of vp : " << vp << endl;
cout << "Address of bobj: " << &bobj << endl;
}
```

The output of this example will be similar to the following. Both `vp` and `&bobj` will refer to the same address:

```
Address of vp : 12FF6C
Address of bobj: 12FF6C
```

The primary purpose for the **dynamic_cast** operator is to perform type-safe *downcasts*. A downcast is the conversion of a pointer or reference to a class A to pointer or reference to a class B, where class A is a base class of B. The problem with downcasts is that a pointer of type A* can and must point to any object of a class that has been derived from A. The **dynamic_cast** operator ensures that if you convert a pointer of class A to a pointer of a class B, the object that A points to belongs to class B or a class derived from B.

The following example demonstrates the use of the **dynamic_cast** operator:

```
#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "Class A" << endl; }
};

struct B : A {
    virtual void f() { cout << "Class B" << endl; }
};

struct C : A {
    virtual void f() { cout << "Class C" << endl; }
};

void f(A* arg) {
    B* bp = dynamic_cast<b>(arg);
    C* cp = dynamic_cast<c>(arg);

    if (bp)
        bp->f();
    else if (cp)
        cp->f();
    else
        arg->f();
};

int main() {
    A aobj;
    C cobj;
    A* ap = &cobj;
    A* ap2 = &aobj;
    f(ap);
    f(ap2);
}
```

The following is the output of the above example:

```
Class C
Class A
```

The function `f()` determines whether the pointer `arg` points to an object of type A, B, or C. The function does this by trying to convert `arg` to a pointer of type B, then to a

pointer of type C, with the **dynamic_cast** operator. If the **dynamic_cast** operator succeeds, it returns a pointer that points to the object denoted by *arg*. If **dynamic_cast** fails, it returns 0.

You may perform downcasts with the **dynamic_cast** operator only on polymorphic classes. In the above example, all the classes are polymorphic because class A has a virtual function. The **dynamic_cast** operator uses the run-time type information generated from polymorphic classes.

►VAC++ You must indicate that you want the compiler to generate run-time type information with a compiler option.

Unary Expressions

A *unary expression* contains one operand and a unary operator. All unary operators have the same precedence and have right- to-left associativity.

As indicated in the following descriptions, the usual arithmetic conversions are performed on the operands of most unary expressions.

The following table summarizes the operators for unary expressions:

Table 5. Precedence and associativity of unary operators

Rank	Right Associative?	Operator Function	Usage
3	yes	size of object in bytes	sizeof (<i>expr</i>)
3	yes	size of type in bytes	sizeof <i>type</i>
3	yes	prefix increment	++ <i>lvalue</i>
3	yes	prefix decrement	-- <i>lvalue</i>
3	yes	complement	~ <i>expr</i>
3	yes	not	! <i>expr</i>
3	yes	unary minus	- <i>expr</i>
3	yes	unary plus	+ <i>expr</i>
3	yes	address of	& <i>lvalue</i>
3	yes	indirection or dereference	* <i>expr</i>
3	yes	► C++ create (allocate memory)	new <i>type</i>
3	yes	► C++ create (allocate and initialize memory)	new <i>type</i> (<i>expr_list</i>) <i>type</i>
3	yes	► C++ create (placement)	new <i>type</i> (<i>expr_list</i>) <i>type</i> (<i>expr_list</i>)
3	yes	► C++ destroy (deallocate memory)	delete <i>pointer</i>
3	yes	► C++ destroy array	delete [] <i>pointer</i>
3	yes	type conversion (cast)	(<i>type</i>) <i>expr</i>

In addition, postfix expressions are also unary expressions.

RELATED REFERENCES

- “Increment ++” on page 114

Unary Expressions

- “Decrement `--`”
- “Unary Plus `+`” on page 115
- “Unary Minus `-`” on page 115
- “Arithmetic Conversions” on page 149

Increment `++`

The `++` (increment) operator adds 1 to the value of a scalar operand, or if the operand is a pointer, increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. The operand must be a modifiable lvalue of arithmetic or pointer type.

You can put the `++` before or after the operand. If it appears before the operand, the operand is incremented. The incremented value is then used in the expression. If you put the `++` after the operand, the value of the operand is used in the expression *before* the operand is incremented. For example:

```
play = ++play1 + play2++;
```

is similar to the following expressions; `play2` is altered before `play`:

```
int temp, temp1, temp2;
```

```
temp1 = play1;
temp2 = play2 + 1;
play1 = play1 + 1;
temp = temp1 + temp2;
play2 = temp2;
play = temp;
```

The result has the same type as the operand after integral promotion.

The usual arithmetic conversions on the operand are performed.

RELATED REFERENCES

- “Pointer Arithmetic” on page 84
- “Lvalues and Rvalues” on page 99
- “Arithmetic Conversions” on page 149

Decrement `--`

The `--` (decrement) operator subtracts 1 from the value of a scalar operand, or if the operand is a pointer, decreases the operand by the size of the object to which it points. The operand receives the result of the decrement operation. The operand must be a modifiable lvalue.

You can put the `--` before or after the operand. If it appears before the operand, the operand is decremented, and the decremented value is used in the expression. If the `--` appears after the operand, the current value of the operand is used in the expression and the operand is decremented.

For example:

```
play = --play1 + play2--;
```

is similar to the following expressions; `play2` is altered before `play`:

```
int temp, temp1, temp2;
```

```
temp1 = play1;
temp2 = play2 - 1;
```

```
play1 = play1 - 1;
temp = temp1 + temp2;
play2 = temp2;
play = temp;
```

The result has the same type as the operand after integral promotion, but is not an lvalue.

The usual arithmetic conversions are performed on the operand.

RELATED REFERENCES

- “Pointer Arithmetic” on page 84
- “Lvalues and Rvalues” on page 99
- “Arithmetic Conversions” on page 149

Unary Plus +

The + (unary plus) operator maintains the value of the operand. The operand can have any arithmetic type or pointer type. The result is not an lvalue.

The result has the same type as the operand after integral promotion.

Note: Any plus sign in front of a constant is not part of the constant.

RELATED REFERENCES

- “Lvalues and Rvalues” on page 99

Unary Minus –

The - (unary minus) operator negates the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

For example, if `quality` has the value 100, `-quality` has the value -100.

The result has the same type as the operand after integral promotion.

Note: Any minus sign in front of a constant is not part of the constant.

RELATED REFERENCES

- “Lvalues and Rvalues” on page 99

Logical Negation !

The ! (logical negation) operator determines whether the operand evaluates to 0 (false) or nonzero (true).

C The expression yields the value 1 (true) if the operand evaluates to 0, and yields the value 0 (false) if the operand evaluates to a nonzero value.

C++ The expression yields the value true if the operand evaluates to false (0), and yields the value false if the operand evaluates to true (nonzero). The operand is implicitly converted to **bool** and the type of the result is **bool**.

The following two expressions are equivalent:

```
!right;
right == 0;
```

Unary Expressions

RELATED REFERENCES

- “Lvalues and Rvalues” on page 99
- “Boolean Variables” on page 46

Bitwise Negation `~`

The `~` (bitwise negation) operator yields the bitwise complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must have an integral type. The result has the same type as the operand but is not an lvalue.

Suppose `x` represents the decimal value 5. The 16-bit binary representation of `x` is:

```
0000000000000101
```

The expression `~x` yields the following result (represented here as a 16-bit binary number):

```
1111111111111010
```

Note that the `~` character can be represented by the trigraph `??~`.

The 16-bit binary representation of `~0` is:

```
1111111111111111
```

RELATED REFERENCES

- “Lvalues and Rvalues” on page 99
- “Trigraph Sequences” on page 15

Address `&`

The `&` (address) operator yields a pointer to its operand. The operand must be an lvalue, a function designator, or a qualified name. It cannot be a bit field, nor can it have the storage class **register**.

C++ You may take the address of a register variable.

If the operand is an lvalue or function, the resulting type is a pointer to the expression type. For example, if the expression has type **int**, the result is a pointer to an object having type **int**.

If the operand is a qualified name and the member is not static, the result is a pointer to a member of class and has the same type as the member. The result is not an lvalue.

If `p_to_y` is defined as a pointer to an **int** and `y` as an **int**, the following expression assigns the address of the variable `y` to the pointer `p_to_y`:

```
p_to_y = &y;
```

C++ You can use the `&` operator with overloaded functions only in an initialization or assignment where the left side uniquely determines which version of the overloaded function is used.

RELATED REFERENCES

- “Lvalues and Rvalues” on page 99
- “Chapter 7. Functions” on page 153

- “Pointers” on page 81
- “Overloading Functions” on page 269
- “register Storage Class Specifier” on page 41

Indirection *

The `*` (indirection) operator determines the value referred to by the pointer-type operand. The operand cannot be a pointer to an incomplete type. The operation yields an lvalue or a function designator if the operand points to a function. Arrays and functions are converted to pointers.

The type of the operand determines the type of the result. For example, if the operand is a pointer to an `int`, the result has type `int`.

Do not apply the indirection operator to any pointer that contains an address that is not valid, such as `NULL`. The result is not defined.

If `p_to_y` is defined as a pointer to an `int` and `y` as an `int`, the expressions:

```
p_to_y = &y;
*p_to_y = 3;
```

cause the variable `y` to receive the value 3.

RELATED REFERENCES

- “Arrays” on page 86
- “Chapter 7. Functions” on page 153
- “Pointers” on page 81

sizeof (Size of an Object)

The **sizeof** operator yields the size in *bytes* of the operand. Types cannot be defined in a **sizeof** expression. The **sizeof** operation cannot be performed on

- A bit field
- A function
- An undefined structure or class
- An incomplete type (such as `void`)

The operand can be the parenthesized name of a type or expression.

The compiler must be able to evaluate the size at compile time. The expression is not evaluated; there are no side effects. For example, the value of `b` is 5 from initialization to the end of program run time:

```
#include <stdio.h>

int main(void){
    int b = 5;
    sizeof(b++);
    return 0;
}
```

The result is an integer constant.

The size of a `char` object is the size of a byte. For example, if a variable `x` has type `char`, the expression `sizeof(x)` always evaluates to 1.

The result of a `sizeof` operation has type `size_t`, which is an unsigned integral type defined in the `<stddef.h>` header.

Unary Expressions

The size of an object is determined on the basis of its definition. The **sizeof** operator does not perform any conversions. If the operand contains operators that perform conversions, the compiler does take these conversions into consideration. The following expression causes the usual arithmetic conversions to be performed. The result of the expression `x + 1` has type **int** (if `x` has type `char`, **short**, or **int** or any enumeration type) and is equivalent to `sizeof(int)`:

```
sizeof (x + 1);
```

Except in preprocessor directives, you can use a **sizeof** expression wherever an integral constant is required. One of the most common uses for the **sizeof** operator is to determine the size of objects that are referred to during storage allocation, input, and output functions.

Another use of **sizeof** is in porting code across platforms. You should use the **sizeof** operator to determine the size that a data type represents. For example:

```
sizeof(int);
```

z/OS Using the `sizeof` operator with `decimal(n,p)` results in the total number of bytes that are occupied by the decimal type. z/OS C/C++ implements decimal data types using the native packed decimal format. Each digit occupies half a byte. The sign occupies an additional half byte. The following example gives you a result of 6 bytes:

```
sizeof(decimal(10,2));
```

The result of a `sizeof` expression depends on the type it is applied to:

- | | |
|--------------------|---|
| An array | The result is the total number of bytes in the array. For example, in an array with 10 elements, the size is equal to 10 times the size of a single element. The compiler does not convert the array to a pointer before evaluating the expression. |
| A class | The result is always nonzero, and is equal to the number of bytes in an object of that class including any padding required for placing class objects in an array. |
| A reference | The result is the size of the referenced object. |

RELATED REFERENCES

- “Integer Constant Expressions” on page 101
- “Arrays” on page 86
- “Chapter 12. Classes” on page 283
- “References” on page 92

digitsof and precisionof (z/OS C Only)

z/OS The `digitsof` and `precisionof` operators yield information about decimal types or an expressions of the decimal type. The `<decimal.h>` header file defines the `digitsof` and `precisionof` macros.

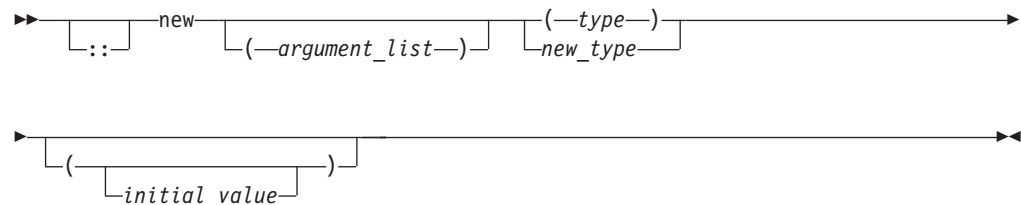
The `digitsof` operator gives the number of significant digits of an object, and `precisionof` gives the number of decimal digits. That is,

```
digitsof(decimal(n,p)) = n  
precisionof(decimal(n,p)) = p
```


The results of the `digitsof` and `precisionof` operators are integer constants. See “Fixed-Point Decimal Constants (z/OS C Only)” on page 27 and “Fixed-Point Decimal Data Types (z/OS C Only)” on page 48 for more information about decimal types.

C++ new Operator

C++ The **new** operator provides dynamic storage allocation. The syntax for an allocation expression containing the **new** operator is:



If you prefix **new** with the scope resolution operator (`::`), the **global operator new()** is used. If you specify an *argument_list*, the overloaded **new** operator that corresponds to that *argument_list* is used. The *type* is an existing built-in or user-defined type. A *new_type* is a type that has not already been defined and can include type specifiers and declarators.

An allocation expression containing the **new** operator is used to find storage in free store for the object being created. The *new expression* returns a pointer to the object created and can be used to initialize the object. If the object is an array, a pointer to the initial element is returned.

You can use **set_new_handler()** only to specify what **new** does when it fails.

You cannot use the **new** operator to allocate function types, **void**, or incomplete class types because these are not object types. However, you can allocate pointers to functions with the **new** operator. You cannot create a reference with the **new** operator.

When the object being created is an array, only the first dimension can be a general expression. All subsequent dimensions must be constant integral expressions. The first dimension can be a general expression even when an existing *type* is used. You can create an array with zero bounds with the **new** operator. For example:

```
char * c = new
char[0];
```

In this case, a pointer to a unique object is returned.

An object created with **operator new()** or **operator new[]()** exists until the **operator delete()** or **operator delete[]()** is called to deallocate the object’s memory. A **delete** operator or a destructor will not be implicitly called for an object created with a **new** that has not been explicitly deallocated before the end of the program.

If parentheses are used within a new type, parentheses should also surround the new type to prevent syntax errors.

In the following example, storage is allocated for an array of pointers to functions:

Unary Expressions

```
void f();
void g();

int main(void)
{
    void (**p)(), (**q)();
    // declare p and q as pointers to pointers to void functions
    p = new (void (*[3])());
    // p now points to an array of pointers to functions
    q = new void(*[3])(); // error
    // error - bound as 'q = (new void) (*[3])();'
    p[0] = f; // p[0] to point to function f
    q[2] = g; // q[2] to point to function g
    p[0](); // call f()
    q[2](); // call g()
    return (0);
}
```

However, the second use of **new** causes an erroneous binding of `q = (new void) (*[3])()`.

The type of the object being created cannot contain class declarations, enumeration declarations, or **const** or **volatile** types. It can contain pointers to **const** or **volatile** objects.

For example, `const char*` is allowed, but `char* const` is not.

Additional arguments can be supplied to **new** by using the *argument_list*, also called the *placement syntax*. If placement arguments are used, a declaration of **operator new()** or **operator new[]()** with these arguments must exist. For example:

```
#include <new>
using namespace std;

class X
{
public:
    void* operator new(size_t,int, int){ /* ... */ }
};


// ...

int main ()
{
    X* ptr = new(1,2) X;
}
```

RELATED REFERENCES

- “C++ Scope Resolution Operator ::” on page 102
- “Free Store” on page 353
- “set_new_handler() — Set Behavior for new Failure” on page 121
- “C++ delete Operator” on page 122
- “Constructors and Destructors Overview” on page 341
- “Objects” on page 34
- “Integer Constant Expressions” on page 101

Initializing Objects Created with the new Operator

 You can initialize objects created with the **new** operator in several ways. For nonclass objects, or for class objects without constructors, a *new initializer* expression can be provided in a new expression by specifying (*expression*) or (). For example:

```
double* pi = new double(3.1415926);
int* score = new int(89);
float* unknown = new float();
```

If a class does not have a default constructor, the `new` initializer must be provided when any object of that class is allocated. The arguments of the `new` initializer must match the arguments of a constructor.


You cannot specify an initializer for arrays. You can initialize an array of class objects only if the class has a default constructor. The constructor is called to initialize each array element (class object).

Initialization using the `new` initializer is performed only if **`new`** successfully allocates storage.

RELATED REFERENCES

- “Free Store” on page 353
- “Constructors and Destructors Overview” on page 341

set_new_handler() — Set Behavior for new Failure

 When the **`new`** operator creates a new object, it calls the **`operator new()`** or **`operator new[]()`** function to obtain the needed storage.

When **`new`** cannot allocate storage to create a new object, it calls a *new handler* function if one has been installed by a call to **`set_new_handler()`**. The **`std::set_new_handler()`** function is declared in the header `<new>`. Use it to call a new handler you have defined or the default new handler.

Your new handler must perform one of the following:

- obtain more storage for memory allocation, then return
- throw an exception of type **`std::bad_alloc`** or a class derived from **`std::bad_alloc`**
- call either **`abort()`** or **`exit()`**

The **`set_new_handler()`** function has the prototype:

```
typedef void(*PNH)();
PNH set_new_handler(PNH);
```

`set_new_handler()` takes as an argument a pointer to a function (the new handler), which has no arguments and returns void. It returns a pointer to the previous new handler function.

If you do not specify your own **`set_new_handler()`** function, **`new`** throws an exception of type **`std::bad_alloc`**.

The following program fragment shows how you could use **`set_new_handler()`** to return a message if the **`new`** operator cannot allocate storage:

```
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

void no_storage()
{
    std::cerr << "Operator new failed: no storage is
    available.\n";
    std::exit(1);
}
```

Unary Expressions

```
}
int main(void)
{
    std::set_new_handler(&no_storage);
    // Rest of program ...
}
```

If the program fails because **new** cannot allocate storage, the program exits with the message:

```
Operator new failed:
no storage is available.
```

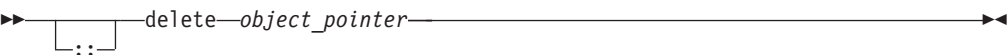
RELATED REFERENCES

- “C++ new Operator” on page 119
- “Free Store” on page 353

C++ delete Operator

C++ The **delete** operator destroys the object created with **new** by deallocating the memory associated with the object.


The **delete** operator has a **void** return type. It has the syntax:

► 

The operand of **delete** must be a pointer returned by **new**, and cannot be a pointer to constant. Deleting a null pointer has no effect.

The **delete[]** operator frees storage allocated for array objects created with **new[]**. The **delete** operator frees storage allocated for individual objects created with **new**.

It has the syntax:

► 

The result of deleting an array object with **delete** is undefined, as is deleting an individual object with **delete[]**. The array dimensions do not need to be specified with **delete[]**.

The result of any attempt to access a deleted object or array is undefined.

If a destructor has been defined for a class, **delete** invokes that destructor. Whether a destructor exists or not, **delete** frees the storage pointed to by calling the function **operator delete()** of the class if one exists.

The global **::operator delete()** is used if:

- The class has no **operator delete()**.
- The object is of a nonclass type.
- The object is deleted with the **::delete** expression.

The global **::operator delete[]()** is used if:

- The class has no **operator delete[]()**
- The object is of a nonclass type
- The object is deleted with the **::delete[]** expression.

The default global **operator delete()** only frees storage allocated by the default global **operator new()**. The default global **operator delete[]()** only frees storage allocated for arrays by the default global **operator new[]()**.

RELATED REFERENCES

- “Free Store” on page 353
- “Constructors and Destructors Overview” on page 341
- “void Type” on page 50

Allocation and Deallocation Functions

You may define your own new operator or allocation function as a class member function or a global namespace function with the following restrictions:

- The first parameter must be of type **std::size_t**. It cannot have a default parameter.
- The return type must be of type **void***.
- Your allocation function may be a template function. Neither the first parameter nor the return type may depend on a template parameter.
- If you declare your allocation function with the empty exception specification **throw()**, your allocation function must return a null pointer your function fails. Otherwise, your function must throw an exception of type **std::bad_alloc** or a class derived from **std::bad_alloc** if your function fails.

You may define your own delete operator or deallocation function as a class member function or a global namespace function with the following restrictions:

- The first parameter must be of type **void***.
- The return type must be of type **void**.
- Your dellocation function may be a template function. Neither the first parameter nor the return type may depend on a template parameter.

The following example defines replacement functions for global namespace **new** and **delete**:

```
#include <cstdio>
#include <cstdlib>

using namespace std;

void* operator new(size_t sz) {
    printf("operator new with %d bytes\n", sz);
    void* p = malloc(sz);
    if (p == 0) printf("Memory error\n");
    return p;
}

void operator delete(void* p) {
    if (p == 0) printf("Deleting a null pointer\n");
    else {
        printf("delete object\n");
        free(p);
    }
}

struct A {
    const char* data;
    A() : data("Text String") { printf("Constructor of S\n"); }
    ~A() { printf("Destructor of S\n"); }
};
```

Unary Expressions

```
int main() {
    A* ap1 = new A;
    delete ap1;

    printf("Array of size 2:\n");
    A* ap2 = new A[2];
    delete[] ap2;
}
```

The following is the output of the above example:

```
operator
new with 40 bytes
operator new with 33 bytes
operator new with 4 bytes
Constructor of S
Destructor of S
delete object
Array of size 2:
operator new with 16 bytes
Constructor of S
Constructor of S
Destructor of S
Destructor of S
delete object
```

RELATED REFERENCES

- “Free Store” on page 353

Binary Expressions

A *binary expression* contains two operands separated by one operator.

Not all binary operators have the same precedence.

All binary operators have left-to-right associativity.

The order in which the operands of most binary operators are evaluated is not specified. To ensure correct results, avoid creating binary expressions that depend on the order in which the compiler evaluates the operands.

As indicated in the following descriptions, the usual arithmetic conversions are performed on the operands of most binary expressions.

The following table summarizes the operators for binary expressions:

Table 6. Precedence and associativity of binary operators

Rank	Right Associative?	Operator Function	Usage
5		multiplication	<i>expr * expr</i>
5		division	<i>expr / expr</i>
5		modulo (remainder)	<i>expr % expr</i>
6		binary addition	<i>expr + expr</i>
6		binary subtraction	<i>expr - expr</i>
7		bitwise shift left	<i>expr << expr</i>
7		bitwise shift right	<i>expr >> expr</i>
8		less than	<i>expr < expr</i>

Table 6. Precedence and associativity of binary operators (continued)

Rank	Right Associative?	Operator Function	Usage
8		less than or equal to	<i>expr <= expr</i>
8		greater than	<i>expr > expr</i>
8		greater than or equal to	<i>expr >= expr</i>
9		equal	<i>expr == expr</i>
9		not equal	<i>expr != expr</i>
10		bitwise AND	<i>expr & expr</i>
11		bitwise exclusive OR	<i>expr ^ expr</i>
12		bitwise inclusive OR	<i>expr expr</i>
13		logical AND	<i>expr && expr</i>
14		logical inclusive OR	<i>expr expr</i>
16	yes	simple assignment	<i>lvalue = expr</i>
16	yes	multiply and assign	<i>lvalue *= expr</i>
16	yes	divide and assign	<i>lvalue /= expr</i>
16	yes	modulo and assign	<i>lvalue %= expr</i>
16	yes	add and assign	<i>lvalue += expr</i>
16	yes	subtract and assign	<i>lvalue -= expr</i>
16	yes	shift left and assign	<i>lvalue <<= expr</i>
16	yes	shift right and assign	<i>lvalue >>= expr</i>
16	yes	bitwise AND and assign	<i>lvalue &= expr</i>
16	yes	bitwise exclusive OR and assign	<i>lvalue ^= expr</i>
16	yes	bitwise inclusive OR and assign	<i>lvalue = expr</i>
18		comma (sequencing)	<i>expr , expr</i>

RELATED REFERENCES

- “Operator Precedence and Associativity” on page 95
- “Arithmetic Conversions” on page 149

Multiplication *

The `*` (multiplication) operator yields the product of its operands. The operands must have an arithmetic or enumeration type. The result is not an lvalue. The usual arithmetic conversions on the operands are performed.

Because the multiplication operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one multiplication operator. For example, the expression:

```
sites * number * cost
```

can be interpreted in any of the following ways:

```
(sites * number) * cost
sites * (number * cost)
(cost * sites) * number
```

RELATED REFERENCES

- “Lvalues and Rvalues” on page 99

Binary Expressions

- “Arithmetic Conversions” on page 149

Division /

The / (division) operator yields the quotient of its operands. The operands must have an arithmetic or enumeration type. The result is not an lvalue.

If both operands are positive integers and the operation produces a remainder, the remainder is ignored. For example, expression `7 / 4` yields the value 1 (rather than 1.75 or 2).

VAC++ **z/OS** On all IBM C and C++ compilers, if either operand is negative, the result is rounded towards zero.

The result is undefined if the second operand evaluates to 0.

The usual arithmetic conversions on the operands are performed.

RELATED REFERENCES

- “Lvalues and Rvalues” on page 99
- “Arithmetic Conversions” on page 149

Remainder %

The % (remainder) operator yields the remainder from the division of the left operand by the right operand. For example, the expression `5 % 3` yields 2. The result is not an lvalue.

Both operands must have an integral or enumeration type. If the right operand evaluates to 0, the result is undefined. If either operand has a negative value, the result is such that the following expression always yields the value of `a` if `b` is not 0 and `a/b` is representable:

$$(a / b) * b + a \% b;$$

The sign of the remainder is the same as the sign of the quotient.

The usual arithmetic conversions on the operands are performed.

RELATED REFERENCES

- “Arithmetic Conversions” on page 149

Addition +

The + (addition) operator yields the sum of its operands. Both operands must have an arithmetic type, or one operand must be a pointer to an object type and the other operand must have an integral or enumeration type.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

A pointer to an object in an array can be added to a value having integral type. The result is a pointer of the same type as the pointer operand. The result refers to another element in the array, offset from the original element by the amount of the integral value treated as a subscript. If the resulting pointer points to storage outside the array, other than the first location outside the array, the result is

undefined. The compiler does not provide boundary checking on the pointers. For example, after the addition, `ptr` points to the third element of the array:

```
int array[5];
int *ptr;
ptr = array + 2;
```

RELATED REFERENCES

- “Pointer Arithmetic” on page 84
- “Pointer Conversions” on page 146

Subtraction –

The `-` (subtraction) operator yields the difference of its operands. Both operands must have an arithmetic or enumeration type, or the left operand must have a pointer type and the right operand must have the same pointer type or an integral or enumeration type. You cannot subtract a pointer from an integral value.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

When the left operand is a pointer and the right operand has an integral type, the compiler converts the value of the right to an address offset. The result is a pointer of the same type as the pointer operand.

If both operands are pointers to the same type, the compiler converts the result to an integral type that represents the number of objects separating the two addresses. Behavior is undefined if the pointers do not refer to objects in the same array.

RELATED REFERENCES

- “Pointer Arithmetic” on page 84
- “Pointer Conversions” on page 146

Bitwise Left and Right Shift << >>

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted. The result is not an lvalue. Both operands have the same precedence and are left-to-right associative.

Operator	Usage
<<	Indicates the bits are to be shifted to the left.
>>	Indicates the bits are to be shifted to the right.

Each operand must have an integral or enumeration type. The compiler performs integral promotions on the operands, and then the right operand is converted to type `int`. The result has the same type as the left operand (after the arithmetic conversions).

The right operand should not have a negative value or a value that is greater than or equal to the width in bits of the expression being shifted. The result of bitwise shifts on such values is unpredictable.

Binary Expressions

If the right operand has the value 0, the result is the value of the left operand (after the usual arithmetic conversions).

The `<<` operator fills vacated bits with zeros. For example, if `left_op` has the value 4019, the bit pattern (in 16-bit format) of `left_op` is:

```
0000111110110011
```

The expression `left_op << 3` yields:

```
0111110110011000
```

Relational `<` `>` `<=` `>=`

The relational operators compare two operands and determine the validity of a relationship.

C The type of the result is **int** and has the values 1 if the specified relationship is true, and 0 if false.

C++ The type of the result is **bool** and has the values **true** or **false**.

The result is not an lvalue.

The following table describes the four relational operators:

Operator	Usage
<code><</code>	Indicates whether the value of the left operand is less than the value of the right operand.
<code>></code>	Indicates whether the value of the left operand is greater than the value of the right operand.
<code><=</code>	Indicates whether the value of the left operand is less than or equal to the value of the right operand.
<code>>=</code>	Indicates whether the value of the left operand is greater than or equal to the value of the right operand.

Both operands must have arithmetic or enumeration types or be pointers to the same type.

C The result has type **int**.

C++ The result has type **bool**.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

When the operands are pointers, the result is determined by the locations of the objects to which the pointers refer. If the pointers do not refer to objects in the same array, the result is not defined.

A pointer can be compared to a constant expression that evaluates to 0. You can also compare a pointer to a pointer of type **void***. The pointer is converted to a pointer of type **void***.

If two pointers refer to the same object, they are considered equal. If two pointers refer to nonstatic members of the same object, the pointer to the object declared later is greater, provided that they are not separated by an access specifier; otherwise the comparison is undefined. If two pointers refer to data members of the same union, they have the same address value.

If two pointers refer to elements of the same array, or to the first element beyond the last element of an array, the pointer to the element with the higher subscript value is greater.

You can only compare members of the same object with relational operators.

Relational operators have left-to-right associativity. For example, the expression:

```
a < b <= c
```

is interpreted as:

```
(a < b) <= c
```

If the value of *a* is less than the value of *b*, the first relationship yields 1 in C, or **true** in C++. The compiler then compares the value **true** (or 1) with the value of *c* (integral promotions are carried out if needed).

Equality == !=

The equality operators, like the relational operators, compare two operands for the validity of a relationship. The equality operators, however, have a lower precedence than the relational operators.

C The type of the result is **int** and has the values 1 if the specified relationship is true, and 0 if false.

C++ The type of the result is **bool** and has the values **true** or **false**.

The following table describes the two equality operators:

Operator	Usage
==	Indicates whether the value of the left operand is equal to the value of the right operand.
!=	Indicates whether the value of the left operand is not equal to the value of the right operand.

Both operands must have arithmetic or enumeration types or be pointers to the same type, or one operand must have a pointer type and the other operand must be a pointer to void or a null pointer. The result is type **int** in C or **bool** in C++.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

If the operands are pointers, the result is determined by the locations of the objects to which the pointers refer.

If one operand is a pointer and the other operand is an integer having the value 0, the == expression is true only if the pointer operand evaluates to NULL. The != operator evaluates to true if the pointer operand does *not* evaluate to NULL.

Binary Expressions

You can also use the equality operators to compare pointers to members that are of the same type but do not belong to the same object. The following expressions contain examples of equality and relational operators:

```
time < max_time == status < complete
letter != EOF
```

Note: The equality operator (==) should not be confused with the assignment (=) operator.

For example,

if (x == 3) evaluates to **true** (or 1) if x is equal to three. Equality tests like this should be coded with spaces between the operator and the operands to prevent unintentional assignments.

while
if (x = 3) is taken to be true because (x = 3) evaluates to a nonzero value (3). The expression also assigns the value 3 to x.

RELATED REFERENCES

- “Simple Assignment =” on page 134

Bitwise AND &

The & (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's, the corresponding bit of the result is set to 1. Otherwise, it sets the corresponding result bit to 0.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands.

Because the bitwise AND operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise AND operator.

The following example shows the values of a, b, and the result of a & b represented as 16-bit binary numbers:

bit pattern of a	0000000001011100
bit pattern of b	000000000101110
bit pattern of a & b	0000000000001100

Note: The bitwise AND (&) should not be confused with the logical AND. (&&) operator. For example,

1 & 4 evaluates to 0
while
1 && 4 evaluates to true

RELATED REFERENCES

- “Logical AND &&” on page 132

Bitwise Exclusive OR ^

The bitwise exclusive OR operator (in EBCDIC, the ^ symbol is represented by the ~ symbol) compares each bit of its first operand to the corresponding bit of the

second operand. If both bits are 1's or both bits are 0's, the corresponding bit of the result is set to 0. Otherwise, it sets the corresponding result bit to 1.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

Because the bitwise exclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise exclusive OR operator. Note that the `^` character can be represented by the trigraph `??^`.

The following example shows the values of `a`, `b`, and the result of `a ^ b` represented as 16-bit binary numbers:

bit pattern of <code>a</code>	0000000001011100
bit pattern of <code>b</code>	000000000101110
bit pattern of <code>a ^ b</code>	0000000001110010

RELATED REFERENCES

- “Trigraph Sequences” on page 15

Bitwise Inclusive OR |

The `|` (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1. If both of the bits are 0, the result of that bit is 0; otherwise, the result is 1.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

Because the bitwise inclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise inclusive OR operator. Note that the `|` character can be represented by the trigraph `??|`.

The following example shows the values of `a`, `b`, and the result of `a | b` represented as 16-bit binary numbers:

bit pattern of <code>a</code>	0000000001011100
bit pattern of <code>b</code>	000000000101110
bit pattern of <code>a b</code>	0000000001111110

Note: The bitwise OR (`|`) should not be confused with the logical OR (`||`) operator. For example,

```
1 | 4 evaluates to 5
while
1 || 4 evaluates to true
```

RELATED REFERENCES

- “Trigraph Sequences” on page 15
- “Logical OR `||`” on page 132

Binary Expressions

Logical AND &&

The && (logical AND) operator indicates whether both operands are true.

C If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0. The type of the result is **int**. Both operands must have a arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

C++ If both operands have values of **true**, the result has the value **true**. Otherwise, the result has the value **false**. Both operands are implicitly converted to **bool** and the result type is **bool**.

Unlike the & (bitwise AND) operator, the && operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to 0 (or **false**), the right operand is not evaluated.

The following examples show how the expressions that contain the logical AND operator are evaluated:

Expression	Result
1 && 0	false or 0
1 && 4	true or 1
0 && 0	false or 0

The following example uses the logical AND operator to avoid division by zero:
(y != 0) && (x / y)

The expression x / y is not evaluated when y != 0 evaluates to 0 (or **false**).

Note: The logical AND (&&) should not be confused with the bitwise AND (&) operator. For example:

```
1 && 4 evaluates to 1 (or true)
while
1 & 4 evaluates to 0
```

RELATED REFERENCES

- “Bitwise AND &” on page 130

Logical OR ||

The || (logical OR) operator indicates whether either operand is true.

C If either of the operands has a nonzero value, the result has the value 1. Otherwise, the result has the value 0. The type of the result is **int**. Both operands must have a arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

C++ If either operand has a value of **true**, the result has the value **true**. Otherwise, the result has the value **false**. Both operands are implicitly converted to **bool** and the result type is **bool**.

Unlike the `|` (bitwise inclusive OR) operator, the `||` operator guarantees left-to-right evaluation of the operands. If the left operand has a nonzero (or **true**) value, the right operand is not evaluated.

The following examples show how expressions that contain the logical OR operator are evaluated:

Expression	Result
<code>1 0</code>	true or 1
<code>1 4</code>	true or 1
<code>0 0</code>	false or 0

The following example uses the logical OR operator to conditionally increment `y`:

```
++x || ++y;
```

The expression `++y` is not evaluated when the expression `++x` evaluates to a nonzero (or **true**) quantity.

Note: The logical OR (`||`) should not be confused with the bitwise OR (`|`) operator. For example:

```
1 || 4 evaluates to 1 (or true)
while
1 | 4 evaluates to 5
```

RELATED REFERENCES

- “Bitwise Inclusive OR `|`” on page 131

C++ Pointer to Member Operators `.*` `->*`

> C++ There are two pointer to member operators: `.*` and `->*`.

The `.*` operator is used to dereference pointers to class members. The first operand must be of class type. If the type of the first operand is class type `T`, or is a class that has been derived from class type `T`, the second operand must be a pointer to a member of a class type `T`.

The `->*` operator is also used to dereference pointers to class members. The first operand must be a pointer to a class type. If the type of the first operand is a pointer to class type `T`, or is a pointer to a class derived from class type `T`, the second operand must be a pointer to a member of class type `T`.

The `.*` and `->*` operators bind the second operand to the first, resulting in an object or function of the type specified by the second operand.

If the result of `.*` or `->*` is a function, you can only use the result as the operand for the `()` (function call) operator. If the second operand is an lvalue, the result of `.*` or `->*` is an lvalue.

RELATED REFERENCES

- “Class Member Lists” on page 293
- “Lvalues and Rvalues” on page 99
- “Pointers to Members” on page 298

Assignment Expressions

Assignment Expressions

An *assignment expression* stores a value in the object designated by the left operand. There are two types of assignment operators: simple assignment and compound assignment.

The left operand in all assignment expressions must be a modifiable lvalue. The type of the expression is the type of the left operand. The value of the expression is the value of the left operand after the assignment has completed.

The result of an assignment expression is not an lvalue.

All assignment operators have the same precedence and have right- to-left associativity.

Simple Assignment =

The simple assignment operator has the following form:

lvalue = *expr*

The operator stores the value of the right operand *expr* in the object designated by the left operand *lvalue*.

The left operand must be a modifiable lvalue. The type of an assignment operation is the type of the left operand.

If the left operand is not a class type, the right operand is implicitly converted to the type of the left operand. This converted type will not be qualified by **const** or **volatile**.

If the left operand is a class type, that type must be complete. The copy assignment operator of the left operand will be called.

If the left operand is an object of reference type, the compiler will assign the value of the right operand to the object denoted by the reference.

z/OS A packed structure or union can be assigned to a nonpacked structure or union of the same type. A nonpacked structure or union can be assigned to a packed structure or union of the same type.

If one operand is packed and the other is not, z/OS C/C++ remaps the layout of the right operand to match the layout of the left. This remapping of structures might degrade performance. For efficiency, when you perform assignment operations with structures or unions, you should ensure that both operands are either packed or nonpacked.

Note: If you assign pointers to structures or unions, the objects they point to must both be either packed or nonpacked. See “Initializing Pointers” on page 82 for more information on assignments with pointers.

RELATED REFERENCES

- “Pointers” on page 81
- “volatile and const Qualifiers” on page 69
- “Pointers to Members” on page 298
- “References” on page 92
- “Structures” on page 51

- “Unions” on page 59
- “Equality == !=” on page 129
- “Initializing Pointers” on page 82

Compound Assignment

The compound assignment operators consist of a binary operator and the simple assignment operator. They perform the operation of the binary operator on both operands and give the result of that operation to the left operand.

The following table shows the operand types of compound assignment expressions:

Operator	Left Operand	Right Operand
<code>+=</code> or <code>-=</code>	Arithmetic	Arithmetic
<code>+=</code> or <code>-=</code>	Pointer	Integral type
<code>*=</code> , <code>/=</code> , and <code>%=</code>	Arithmetic	Arithmetic
<code><<=</code> , <code>>>=</code> , <code>&=</code> , <code>^=</code> , and <code> =</code>	Integral type	Integral type

Note that the expression

```
a *= b + c
```

is equivalent to

```
a = a * (b + c)
```

and *not*

```
a = a * b + c
```

The following table lists the compound assignment operators and shows an expression using each operator:

Operator	Example	Equivalent Expression
<code>+=</code>	<code>index += 2</code>	<code>index = index + 2</code>
<code>-=</code>	<code>*(pointer++) -= 1</code>	<code>*pointer = *(pointer++) - 1</code>
<code>*=</code>	<code>bonus *= increase</code>	<code>bonus = bonus * increase</code>
<code>/=</code>	<code>time /= hours</code>	<code>time = time / hours</code>
<code>%=</code>	<code>allowance %= 1000</code>	<code>allowance = allowance % 1000</code>
<code><<=</code>	<code>result <<= num</code>	<code>result = result << num</code>
<code>>>=</code>	<code>form >>= 1</code>	<code>form = form >> 1</code>
<code>&=</code>	<code>mask &= 2</code>	<code>mask = mask & 2</code>
<code>^=</code>	<code>test ^= pre_test</code>	<code>test = test ^ pre_test</code>
<code> =</code>	<code>flag = ON</code>	<code>flag = flag ON</code>

Although the equivalent expression column shows the left operands (from the example column) twice, it is in effect evaluated only once.

Cast Expressions

The cast operator is used for *explicit type conversions*. This operator has the following form, where *T* is a type, and *expr* is an expression:

```
( T ) expr
```

Cast Expressions

It converts the value of *expr* to the type *T*. The result of this operation is an lvalue if *T* is a reference. In all other cases, the result is an rvalue.

► C++ You can also use the following function-style notation:

expr(*T*)

This form also converts the value of *expr* to the type *T*. A function-style cast with no arguments, such as *X*() is equivalent to the declaration *X t*(), where *t* is a temporary object. Similarly, a function-style cast with more than one argument, such as *X*(*a*, *b*), is equivalent to the declaration *X t*(*a*, *b*).

► C++ For C++, the operand can have class type. If the operand has class type, it can be cast to any type for which the class has a user-defined conversion function. Casts can invoke a constructor, if the target type is a class, or they can invoke a conversion function, if the source type is a class. They can be ambiguous if both conditions hold.

► C++ An explicit type conversion can also be expressed by using the C++ type conversion operator **static_cast**.

The following demonstrates the use of the cast operator. The example dynamically creates an integer array of size 10:

```
#include <stdlib.h>

int main(void) {
    int* myArray = (int*) malloc(10 * sizeof(int));
    free(myArray);
    return 0;
}
```

The `malloc()` library function returns a **void** pointer that points to memory that will hold an object of the size of its argument. The statement `int* myArray = (int*) malloc(10 * sizeof(int))` does the following

- Creates a void pointer that points to memory that can hold ten integers.
- Converts that **void** pointer into an integer pointer with the use of the cast operator.
- Assigns that integer pointer to `myArray`. Because a name of an array is the same as a pointer to the initial element of the array, `myArray` is an array of ten integers stored in the memory created by the call to `malloc()`.

RELATED REFERENCES

- “Conversion Functions” on page 361
- “Conversion by Constructor” on page 360
- “Standard Type Conversions” on page 144
- “Lvalues and Rvalues” on page 99
- “References” on page 92
- “Temporary Objects” on page 357

C++ throw Expressions

► **C++** A *throw expression* is used to throw exceptions to C++ exception handlers. A throw expression is of type **void**.

RELATED REFERENCES

- “Chapter 17. Exception Handling” on page 401
- “void Type” on page 50

Conditional Expressions

A *conditional expression* is a compound expression that contains a condition implicitly converted to bool (*operand₁*), an expression to be evaluated if the condition evaluates to true (*operand₂*), and an expression to be evaluated if the condition has the value false (*operand₃*).

Conditional expressions have right-to-left associativity. The left most operand is evaluated first, and then only one of the remaining two operands is evaluated.

The conditional expression contains one two-part operator. The ? symbol follows the condition, and the : symbol appears between the two action expressions. All expressions that occur between the ? and : are treated as one expression.

The first operand must have a scalar type. The type of the second and third operands must be one of the following:

- An arithmetic type
- A compatible pointer, structure, or union type
- void

The second and third operands can also be a pointer or a null pointer constant.

Two objects are compatible when they have the same type but not necessarily the same type qualifiers (**volatile** or **const**). Pointer objects are compatible if they have the same type or are pointers to void.

The first operand is evaluated, and its value determines whether the second or third operand is evaluated:

- If the value is true, the second operand is evaluated.
- If the value is false, the third operand is evaluated.

The result is the value of the second or third operand.

If the second and third expressions evaluate to arithmetic types, the usual arithmetic conversions are performed on the values. The types of the second and third operands determine the type of the result as shown in the following tables.

Type of Conditional C Expressions

► **C**

Type of One Operand	Type of Other Operand	Type of Result
Arithmetic	Arithmetic	Arithmetic type after usual arithmetic conversions

Conditional Expressions

Type of One Operand	Type of Other Operand	Type of Result
Structure or union type	Compatible structure or union type	Structure or union type with all the qualifiers on both operands
void	void	void
Pointer to compatible type	Pointer to compatible type	Pointer to type with all the qualifiers specified for the type
Pointer to type	NULL pointer (the constant 0)	Pointer to type
Pointer to object or incomplete type	Pointer to void	Pointer to void with all the qualifiers specified for the type

Type of Conditional C++ Expressions

➤ C++

Type of One Operand	Type of Other Operand	Type of Result
Reference to type	Reference to type	Reference after usual reference conversions
Class T	Class T	Class T
Class T	Class X	Class type for which a conversion exists. If more than one possible conversion exists, the result is ambiguous.
throw expression	Other (type, pointer, reference)	Type of the expression that is not a throw expression

Examples of Conditional Expressions

The following expression determines which variable has the greater value, y or z, and assigns the greater value to the variable x:

```
x = (y > z) ? y : z;
```

The following is an equivalent statement:

```
if (y > z)
    x = y;
else
    x = z;
```

The following expression calls the function `printf`, which receives the value of the variable `c`, if `c` evaluates to a digit. Otherwise, `printf` receives the character constant `'x'`.

```
printf(" c = %c\n", isdigit(c) ? c : 'x');
```

If the last operand of a conditional expression contains an assignment operator, use parentheses to ensure the expression evaluates properly. For example, the `=` operator has higher precedence than the `?:` operator in the following expression:

```
int i,j,k;
(i == 7) ? j ++ : k = j;
```

The compiler will interpret this expression as if it were parenthesized this way:

```
int i,j,k;
((i == 7) ? j ++ : k) = j;
```

That is, *k* is treated as the third operand, not the entire assignment expression *k* = *j*.

To assign the value of *j* to *k* *i* == 7 is false, enclose the last operand in parentheses:

```
int i,j,k;
(i == 7) ? j ++ : (k = j);
```

The typeid Operator

The *typeid* operator provides program variable and expression type information. The operator has the following form:

typeid (*expr*)

The **typeid** operator returns an lvalue of type **const std::type_info** that represents the type of expression *expr*.

You must include the standard template library header **<typeinfo>** to use the *typeid* operator.

If *expr* is a reference or a dereferenced pointer to a polymorphic class, **typeid** will return a **type_info** object that represents the object that the reference or pointer denotes at run time. If it is not a polymorphic class, **typeid** will return a **type_info** object that represents the type of the reference or dereferenced pointer. The following example demonstrates this:

```
#include <iostream>
#include <typeinfo>
using namespace std;

struct A { virtual ~A() { } };
struct B : A { };

struct C { };
struct D : C { };

int main() {
    B bobj;
    A* ap = &bobj;
    A& ar = bobj;
    cout << "ap: " << typeid(*ap).name() << endl;
    cout << "ar: " << typeid(ar).name() << endl;

    D dobj;
    C* cp = &dobj;
    C& cr = dobj;
    cout << "cp: " << typeid(*cp).name() << endl;
    cout << "cr: " << typeid(cr).name() << endl;
}
```

The following is the output of the above example:

```
ap: B
ar: B
cp: C
cr: C
```

Conditional Expressions

Classes A and B are polymorphic; classes C and D are not. Although `cp` and `cr` refer to an object of type D, `typeid(*cp)` and `typeid(cr)` return objects that represent class C.

Lvalue-to-rvalue, array-to-pointer, and function-to-pointer conversions will not be applied to *expr*. For example, the output of the following example will be `int [10]`, not `int *`:

```
#include <iostream>
#include <typeinfo>
using namespace std;

int main() {
    int myArray[10];
    cout << typeid(myArray).name() << endl;
}
```

If *expr* is a class type, that class must be completely defined.

The **typeid** operator ignores top-level **const** or **volatile** qualifiers.

Comma Expression ,

A *comma expression* contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. The left operand is evaluated, possibly producing side effects, and the value is discarded. The result of a comma expression is not an lvalue.

Both operands of a comma expression can have any type. All comma expressions have left-to-right associativity. The left operand is fully evaluated before the right operand.

In the following example, if `omega` has the value 11, the expression increments `delta` and assigns the value 3 to `alpha`:

```
alpha = (delta++, omega % 4);
```

Any number of expressions separated by commas can form a single expression. The compiler evaluates the left-most expression first. The value of the right-most expression becomes the value of the entire expression.

For example, the value of the expression:

```
intensity++, shade * increment, rotate(direction);
```

is the value of the expression:

```
rotate(direction)
```

The primary use of the comma operator is to produce side effects in the following situations:

- Calling a function
- Entering or repeating an iteration loop
- Testing a condition
- Other situations where a side effect is required but the result of the expression is not immediately needed

To use the comma operator in a context where the comma has other meanings, such as in a list of function arguments or a list of initializers, you must enclose the comma operator in parentheses. For example, the function

```
f(a, (t = 3, t + 2), c);
```

has only three arguments: the value of `a`, the value 5, and the value of `c`. The value of the second argument is the result of the comma expression in parentheses:

```
t = 3, t + 2
```

which has the value 5.

The following table gives some examples of the uses of the comma operator:

Statement	Effects
<code>for (i=0; i<2; ++i, f());</code>	A for statement in which <code>i</code> is incremented and <code>f()</code> is called at each iteration.
<code>if (f(), ++i, i>1) { /* ... */ }</code>	An if statement in which function <code>f()</code> is called, variable <code>i</code> is incremented, and variable <code>i</code> is tested against a value. The first two expressions within this comma expression are evaluated before the expression <code>i>1</code> . Regardless of the results of the first two expressions, the third is evaluated and its result determines whether the if statement is processed.
<code>func((++a, f(a)));</code>	A function call to <code>func()</code> in which <code>a</code> is incremented, the resulting value is passed to a function <code>f()</code> , and the return value of <code>f()</code> is passed to <code>func()</code> . The function <code>func()</code> is passed only a single argument, because the comma expression is enclosed in parentheses within the function argument list.

Comma Expression

Chapter 6. Implicit Type Conversions

An expression *e* of a given type is *implicitly converted* if used in one of the following situations:

- Expression *e* is used as an operand of an arithmetic or logical operation.
- Expression *e* is used as a condition in an **if** statement or an iteration statement (such as a **for** loop). Expression *e* will be converted to **bool** (or **int** in C).
- Expression *e* is used in a **switch** statement. Expression *e* will be converted to an integral type.
- Expression *e* is used in an initialization. This includes the following:
 - An assignment is made to an lvalue that has a different type than *e*.
 - A function is provided an argument value of *e* that has a different type than the parameter.
 - Expression *e* is specified in the **return** statement of a function, and *e* has a different type from the defined return type for the function.

The compiler will allow an implicit conversion of an expression *e* to a type *T* if and only if the compiler would allow the following statement:

```
T var = e;
```

For example when you add values having different data types, both values are first converted to the same type: when a **short int** value and an **int** value are added together, the **short int** value is converted to the **int** type.


You can perform explicit type conversions using one of the cast operators, the function style cast, or the C style cast.

RELATED REFERENCES

- “Chapter 5. Expressions and Operators” on page 95
- “static_cast Operator” on page 108
- “reinterpret_cast Operator” on page 109
- “const_cast Operator” on page 110
- “dynamic_cast Operator” on page 111
- “Cast Expressions” on page 135

Integral and Floating-Point Promotions

An *integral promotion* is the conversion of one integral type to another where the second type can hold all possible values of the first type. Certain fundamental types can be used wherever an integer can be used. The following fundamental types can be converted through integral promotion are:

- **char**
-  **bool**
- **wchar_t**
- **short int**
- enumerators
- objects of enumeration type
- integer bit fields (both signed and unsigned)

Except for **wchar_t**, if the value cannot be represented by an **int**, the value is converted to an **unsigned int**. For **wchar_t**, if an **int** can represent all the values of the original type, the value is converted to the type that can best represent all the values of the original type. For example, if a **long** can represent all the values, the value is converted to a **long**.

Integral Promotions

Floating-Point Promotions You can convert an rvalue of type **float** to an rvalue of type **double**. The value of the expression is unchanged. This conversion is a *floating-point promotion*.

RELATED REFERENCES

- “char and wchar_t Type Specifiers” on page 45
- “Boolean Variables” on page 46
- “Integer Variables” on page 49
- “Enumerations” on page 65
- “Declaring and Using Bit Fields in Structures” on page 55

Standard Type Conversions

Many C and C++ operators cause *implicit type conversions*, which change the type of an expression. When you add values having different data types, both values are first converted to the same type. For example, when a **short int** value and an **int** value are added together, the **short int** value is converted to the **int** type. It can result in loss of data if the value of the original object is outside the range representable by the shorter type.

Implicit type conversions can occur when:

- An operand is prepared for an arithmetic or logical operation.
- An assignment is made to an lvalue that has a different type than the assigned value.
- A function is provided an argument value that has a different type than the parameter.
- The value specified in the **return** statement of a function has a different type from the defined return type for the function.

You can perform explicit type conversions using the C-style cast, the C++ function-style cast, or one of the C++ cast operators.

```
#include <iostream>
using namespace std;

int main() {
    float num = 98.76;
    int x1 = (int) num;
    int x2 = int(num);
    int x3 = static_cast<int>(num);

    cout << "x1 = " << x1 << endl;
    cout << "x2 = " << x2 << endl;
    cout << "x3 = " << x3 << endl;
}
```

The following is the output of the above example:

```
x1 = 98
x2 = 98
x3 = 98
```

The integer x1 is assigned a value in which num has been explicitly converted to an **int** with the C- style cast. The integer x2 is assigned a value that has been converted with the function-style cast. The integer x3 is assigned a value that has been converted with the **static_cast** operator.

RELATED REFERENCES

- “return Statement” on page 192

- “Cast Expressions” on page 135

Lvalue-to-Rvalue Conversions

If an lvalue appears in a situation in which the compiler expects an rvalue, the compiler converts the lvalue to an rvalue.

An lvalue *e* of a type *T* can be converted to an rvalue if *T* is not a function or array type. The type of *e* after conversion will be *T*. The following table lists exceptions to this:

Situation before conversion	Resulting behavior
<i>T</i> is an incomplete type	compile-time error
<i>e</i> refers to an uninitialized object	undefined behavior
<i>e</i> refers to an object not of type <i>T</i> , nor a type derived from <i>T</i>	undefined behavior
<i>T</i> is a non-class type	the type of <i>e</i> after conversion is <i>T</i> , not qualified by either const or volatile

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69

Boolean Conversions

C++ You can convert integral, floating-point, arithmetic, enumeration, pointer, and pointer to member rvalue types to an rvalue of type **bool**. Any value other than a zero, null pointer, or null member pointer value is converted to **true**; A zero, null pointer, or null member pointer value is converted to **false**.

The following is an example of boolean conversions:

```
void f(int* a, int b)
{
    bool d = a; // false if a == NULL
    bool e = b; // false if b == 0
}
```

The variable *d* will be **false** if *a* is equal to a null pointer. Otherwise, *d* will be **true**. The variable *e* will be **false** if *b* is equal to zero. Otherwise, *e* will be **true**.

RELATED REFERENCES

- “Boolean Variables” on page 46

Integral Conversions

You can convert the following:

- An rvalue of integer type (including signed and unsigned integer types) to another rvalue of integer type
- An rvalue of enumeration type to an rvalue of integer type

If you are converting an integer *a* to an unsigned type, the resulting value *x* is the least unsigned integer such that *a* and *x* are congruent modulo 2^n , where *n* is the number of bits used to represent an unsigned type. If two numbers *a* and *x* are congruent modulo 2^n , the following expression is true, where the function *pow*(*m*, *n*) returns the value of *m* to the power of *n*:

```
a % pow(2, n) == x % pow(2, n)
```

Standard Type Conversions

If you are converting an integer `a` to a signed type, the compiler does not change the resulting value if the new type is large enough to hold the `a`. If the new type is not large enough, the behavior is defined by the compiler.

If you are converting a **bool** to an integer, values of **false** are converted to 0; values of **true** are converted to 1.

Integer promotions belong to a different category of conversions; they are not integral conversions.

RELATED REFERENCES

- “Integer Variables” on page 49

Signed-Integer Conversions (z/OS)

z/OS The z/OS C/C++ compiler converts a signed integer to a shorter integer by truncating the high-order bits. It converts an integer to a longer signed integer by sign-extension.

When converting an integral type to a floating-point type, if the value cannot be represented precisely, HEX floating-point mode will round the value to the nearest representable number, whereas IEEE mode will round according to the specified rounding mode.

For IEEE floating-point types, the rounding mode for static initializers (compile-time rounding) is controlled by the `ROUND` compiler option. The rounding mode during execution time is controlled by the `fp_swap_rnd` run-time function.

When converting a signed integer to an unsigned integer, z/OS C/C++ converts the signed integer to the size of the unsigned integer. It interprets the result as an unsigned value.

When converting a **long long int** type to packed decimal, the resulting size is `decimal(20,0)`.

Floating-Point Conversions

You can convert an rvalue of floating-point type to another rvalue of floating-point type.

Floating-point promotions (converting from **float** to **double**) belong to a different category of conversions; they are not floating-point conversions.

RELATED REFERENCES

- “Floating-Point Variables” on page 47
- “Integral and Floating-Point Promotions” on page 143

Pointer Conversions

Pointer conversions are performed when pointers are used, including pointer assignment, initialization, and comparison.

Conversion to void*

Standard Type Conversions

Any pointer to an object of a type *T*, optionally qualified with **const**, **volatile**, or **const volatile**, can be converted to **void***, keeping the same **const** or **volatile** qualifications. You can also convert any pointer to a function to a **void***, provided that a **void*** has sufficient bits to hold it.

Derived-to-Base Conversions

You can convert an rvalue pointer of type *B** to an rvalue pointer of class *A** where *A* is an accessible base class of *B* as long as the conversion is not ambiguous. The conversion is ambiguous if the expression for the accessible base class can refer to more than one distinct class. The resulting value points to the base class subobject of the derived class object. If the pointer of type *B** is null, it will be converted to a null pointer of type *A**. Note that you cannot convert a pointer to a class into a pointer to its base class if the base class is a virtual base class of the derived class.

Null Pointer Constants

A constant expression that evaluates to zero is a *null pointer constant*. This expression can be converted to a pointer. This pointer will be a null pointer (pointer with a zero value), and is guaranteed not to point to any object.

Array-to-Pointer Conversions

You can convert an lvalue or rvalue with type "array of *N*," where *N* is the type of a single element of the array, to *N**. The result is a pointer to the initial element of the array. You cannot perform the conversion if the expression is used as the operand of the **&** (address) operator or the **sizeof** operator.

Function-to-Pointer Conversions

You can convert an lvalue that is a function of type *T* to an rvalue that is a pointer to a function of type *T*, except when the expression is used as the operand of the **&** (address) operator, the **()** (function call) operator, or the **sizeof** operator.

RELATED REFERENCES

- "void Type" on page 50
- "Pointers" on page 81
- "Integer Constant Expressions" on page 101
- "Arrays" on page 86
- "Pointers to Functions" on page 173

Reference Conversions

A reference conversion can be performed wherever a reference initialization occurs, including reference initialization done in argument passing and function return values. A reference to a class can be converted to a reference to an accessible base class of that class as long as the conversion is not ambiguous. The result of the conversion is a reference to the base class subobject of the derived class object.

Reference conversion is allowed if the corresponding pointer conversion is allowed.

RELATED REFERENCES

- "References" on page 92
- "Initializing References" on page 93
- "Calling Functions and Passing Arguments" on page 164

Standard Type Conversions

- “Function Return Values” on page 171

Pointer-to-Member Conversions

Pointer-to-member conversion can occur when pointers to members are initialized, assigned, or compared.

A constant expression that evaluates to zero can be converted to the null pointer to a member.

Note: A pointer to a member is not the same as a pointer to an object or a pointer to a function.

A pointer to a member of a base class can be converted to a pointer to a member of a derived class if the following conditions are true:

- The conversion is not ambiguous. The conversion is ambiguous if multiple instances of the base class are in the derived class.
- A pointer to the derived class can be converted to a pointer to the base class. If this is the case, the base class is said to be *accessible*.
- Member types must match. For example suppose class A is a base class of class B. You cannot convert a pointer to member of A of type **int** to a pointer to member of type B of type **float**.
- The base class cannot be virtual.

RELATED REFERENCES

- “Integer Constant Expressions” on page 101
- “Access Control of Base Class Members” on page 321
- “Pointers to Members” on page 298
- “C++ Pointer to Member Operators . * ->*” on page 133

Qualification Conversions

You can convert an rvalue of type *cv1* T* where *cv1* is any combination of zero or more **const** or **volatile** qualifications, to an rvalue of type *cv2* T* if *cv2* T* is more **const** or **volatile** qualified than *cv1* T*.

You can convert an rvalue of type pointer to member of a class X of *cv1* T, to an rvalue of type pointer to member of a class X of *cv2* T if *cv2* T is more **const** or **volatile** qualified than *cv1* T.

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69

Function Argument Conversions

C If a function declaration is present and includes declared argument types, the compiler performs type checking. If no function declaration is visible when a function is called, the compiler can perform default argument promotions, which consist of the following:

- Integral promotions
- Arguments with type **float** are converted to type **double**.

C++ Function declarations in C++ must always specify their parameter types. Also, functions may not be called if it has not already been declared.

RELATED REFERENCES

- “Integral and Floating-Point Promotions” on page 143
- “Function Declarations” on page 154

Other Conversions

By definition, the **void** type has no value. Therefore, it cannot be converted to any other type, and no other value can be converted to **void** by assignment. However, a value can be explicitly cast to **void**.

No conversions between structure or union types are allowed.

There are no standard conversions between class types.

C++ You can write your own conversion operators for class types.

C In C, when you define a value using the **enum** type specifier, the value is treated as an **int**. Conversions to and from an **enum** value proceed as for the **int** type.

You can convert from an **enum** to any integral type but not from an integral type to an **enum**.

z/OS When a packed decimal type is converted to a **long long int** type, z/OS C/C++ discards the fractional part.

RELATED REFERENCES

- “void Type” on page 50
- “User-Defined Conversions” on page 358
- “Enumerations” on page 65

Arithmetic Conversions

The conversions depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integer and floating-point types. These standard conversions are known as the *arithmetic conversions* because they apply to the types of values ordinarily used in arithmetic.

Arithmetic conversions are used for matching operands of arithmetic operators.

Arithmetic conversion proceeds in the following order:

Operand Type	Conversion
One operand has long double type	The other operand is converted to long double .
One operand has double type	The other operand is converted to double .
One operand has float type	The other operand is converted to float .
One operand has unsigned long long int type	The other operand is converted to unsigned long long int .
One operand has long long type.	The other operand is converted to long long .
One operand has unsigned long int type	The other operand is converted to unsigned long int .

Arithmetic Conversions

Operand Type	Conversion
One operand has unsigned int type and the other operand has long int type and the value of the unsigned int can be represented in a long int	The operand with unsigned int type is converted to long int .
One operand has unsigned int type and the other operand has long int type and the value of the unsigned int cannot be represented in a long int	Both operands are converted to unsigned long int .
One operand has long int type	The other operand is converted to long int .
One operand has unsigned int type	The other operand is converted to unsigned int .
Both operands have int type	The result is type int .

RELATED REFERENCES

- “Chapter 5. Expressions and Operators” on page 95
- “Integer Variables” on page 49
- “Floating-Point Variables” on page 47

The explicit Keyword

C++ The *explicit* keyword controls unwanted implicit type conversions. It can only be used in declarations of constructors within a class declaration.

A constructor declared with only one argument and without the *explicit* keyword is a *converting constructor*. You can construct objects with a converting constructor using the assignment operator. Declaring a constructor of this type with the *explicit* keyword prevents this behavior. For example, except for the default constructor, the constructors in the following class are converting constructors.

```
class A
{
public:
    A();
    A(int);
    A(const char*, int = 0);
};
```

The following declarations are legal.

```
A c = 1;
A d = "Venditti";
```

The first declaration is equivalent to `A c = A(1)`.

If you declare the constructor of the class with the *explicit* keyword, the previous declarations would be illegal.

For example, if you declare the class as:

```
class A
{
public:
    explicit A();
    explicit A(int);
    explicit A(const char*, int = 0);
};
```

You can only assign values that match the values of the class type.

For example, the following statements will be legal:

```
A a1;  
A a2 = A(1);  
A a3(1);  
A a4 = A("Venditti");  
A* p = new A(1);  
A a5 = (A)1;  
A a6 = static_cast<A>(1);
```

Arithmetic Conversions

Chapter 7. Functions

A function *declaration* consists of a return type, a name, and an argument list. It is used to declare the format and existence of a function prior to its use.

A function *definition* contains a function declaration and the body of the function. A function can only have one definition.

The declaration is used by the compiler for argument type checking and argument conversions. Declarations can appear several times in a program, provided the declarations are compatible. They allow the compiler to check for mismatches between the parameters of a function call and those in the function declaration.

Declarations are typically placed in header files, while function definitions appear in source files.

RELATED REFERENCES

- “Function Declarations” on page 154
- “Function Definitions” on page 158

C++ Enhancements to C Functions

C++ The C++ language provides many enhancements to C functions. These are:

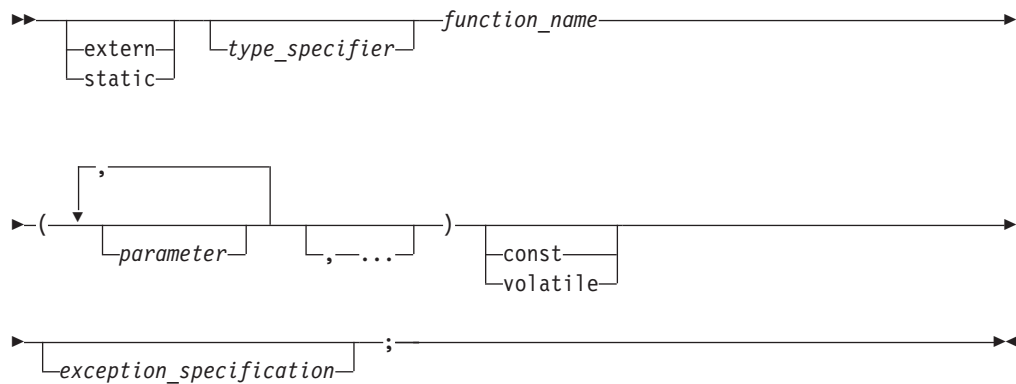
- Reference arguments
- Default arguments
- Reference return types
- Inline functions
- Member functions
- Overloaded functions
- Operator functions
- Constructor and destructor functions
- Conversion functions
- Virtual functions
- Function templates
- Exception specifications
- Constructor initializers

RELATED REFERENCES

- “Passing Arguments by Reference” on page 167
- “Default Arguments in C++ Functions” on page 169
- “Using References as Return Types” on page 172
- “Inline Functions” on page 174
- “Member Functions” on page 295
- “Overloading Functions” on page 269
- “Overloading Operators” on page 271
- “Constructors and Destructors Overview” on page 341
- “Conversion Functions” on page 361
- “Virtual Functions” on page 333

Function Declarations

A function declaration establishes the name and the number and types of the parameters of the function.



A *function argument* is an expression that you use within the parenthesis of a function call. A *function parameter* is an object or reference declared within the parenthesis of a function declaration or definition. When you call a function, the arguments are evaluated, and each parameter is initialized with the value of the corresponding argument. The semantics of argument passing are identical to those of assignment.

Implicit declaration of functions is not allowed.

► **C** The default return type of a function is **int**.

► **C++** There is no default return type.

To indicate that the function does not return a value, declare it with a return type of **void**.

► **C++** Only member functions may have **const** or **volatile** specifiers after the parenthesized parameter list.

A function cannot be declared as returning a data object having a **volatile** or **const** type but it can return a pointer to a **volatile** or **const** object. A function may return a pointer to function, or a pointer to the first element of an array, but it may not return a value that has a type of array or function.

► **C++** The *exception_specification* limits the function from throwing only a specified list of exceptions.

Some declarations do not name the parameters within the parameter lists; the declarations simply specify the types of parameters and the return values. This is called *prototyping*. The following example demonstrates this:

```
int
func(int, long);
```

C **C++** The ellipsis (...) may be the only argument in C++. In this case, the comma is not required. In C, you cannot have the ellipsis as the only argument.

Types cannot be defined in return or argument types. For example, the C++ compiler will allow the following declaration of `print()`:

```
struct X { int i; };
void print(X x);
```

Similarly, the C compiler will allow the following declaration:

```
struct X { int i; };
void print(struct X x);
```

The C and C++ compilers will not allow the following declaration of the same function:

```
void print(struct X { int i; } x); //error
```

This example attempts to declare a function `print()` that takes an object `x` of class `X` as its argument. However, the class definition is not allowed within the argument list.

In another example, the C++ compiler will allow the following declaration of `counter()`:

```
enum count {one, two, three};
count counter();
```

Similarly the C compiler will allow the following declaration:

```
enum count {one, two, three};
enum count counter();
```

The C and C++ compilers will not allow the following declaration of the same function:

```
enum count{one, two, three} counter(); //error
```

In the attempt to declare `counter()`, the enumeration type definition cannot appear in the return type of the function declaration.

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69
- “Exception Specifications” on page 412

C++ Function Declarations

C++ In C++, you can specify the qualifiers **volatile** and **const** in member function declarations. You can also specify exception specifications in function declarations. All C++ functions must be declared before they can be called.

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69
- “const and volatile Member Functions” on page 296
- “Exception Specifications” on page 412

Multiple Function Declarations

C++ All function declarations for one particular function must have the same number and type of parameters, and must have the same return type.

Function Declarations

These return and parameter types are part of the function type, although the default arguments and exception specifications are not.

If a previous declaration of an object or function is visible in an enclosing scope, the identifier has the same linkage as the first declaration. For example, a variable or function that is first declared with the keyword **static** and later declared with the keyword **extern** has internal linkage. However, a variable or function that has no linkage and later declared with a linkage specifier will have the linkage you have specified.

For the purposes of argument matching, ellipsis and linkage keywords are considered a part of the function type. They must be used consistently in all declarations of a function. If the only difference between the parameter types in two declarations is in the use of **typedef** names or unspecified argument array bounds, the declarations are the same. A **const** or **volatile** specifier is also part of the function type, but can only be part of a declaration or definition of a nonstatic member function.

You may *overload* function names. An overloaded function declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different types. If you call an overloaded function name, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions.


Declaring two functions differing only in return type is not valid function overloading, and is flagged as an error. For example:

```
void f();  
int f();           // error, two definitions differ only in  
                  // return type  
  
int g()  
{  
    return f();  
}
```

RELATED REFERENCES

- “Overloading Functions” on page 269

Argument Names in Function Declarations

 You can supply parameter names in a function declaration, but the compiler ignores them except in the following two situations:

1. If two parameter names have the same name within a single declaration. This is an error.
2. If a parameter name is the same as a name outside the function. In this case the name outside the function is hidden and cannot be used in the parameter declaration.

In the following example, the third parameter name `intersects` is meant to have enumeration type `subway_line`, but this name is hidden by the name of the first parameter. The declaration of the function `subway()` causes a compile-time error because `subway_line` is not a valid type name because the first parameter name `subway_line` hides the namespace scope **enum** type and cannot be used again in the second parameter.

```
enum subway_line {yonge,
university, spadina, bloor};
int subway(char * subway_line, int stations,
           subway_line intersects);
```

RELATED REFERENCES

- “Function Declarations” on page 154

Examples of Function Declarations

The following code fragments show several function declarations. The first declares a function `f` that takes two integer arguments and has a return type of **void**:

```
void f(int, int);
```

The following code fragment declares a pointer `p1` to a function that takes a pointer to a constant character and returns an integer:

```
int (*p1) (const char*);
```

The following code fragment declares a function `f1` that takes an integer argument, and returns a pointer to a function that takes an integer argument and returns an integer:

```
int (*f1(int)) (int);
```

Alternatively, a **typedef** can be used for the complicated return type of function `f1`:

```
typedef int f1_return_type(int);
f1_return_type* f1(int);
```

► **C++** The following declaration is of an external function `f2` that takes a constant integer as its first argument, can have a variable number and variable types of other arguments, and returns type **int**.

```
int extern f2(const int ...);
```

► **C** In C, a comma is required before the ellipsis:

```
int extern f2(const int, ...);
```

Function `f3` has a return type **int**, and takes a **int** argument with a default value that is the value returned from function `f2`:

```
const int j = 5;
int f3( int x = f2(j) );
```

► **C++** Function `f6` is a **const** class member function of class `X`, takes no arguments, and has a return type of **int**:

```
class X
{
public:
    int f6() const;
};
```

► **C++** Function `f4` takes no arguments, has return type **void**, and can throw class objects of types `X` and `Y`.

Function Declarations

```
class X;  
class Y;  
  
// ...  
  
void f4() throw(X,Y);
```

► **C++** Function f5 takes no arguments, has return type **void**, and will call **unexpected()** if it throws an exception of any type.

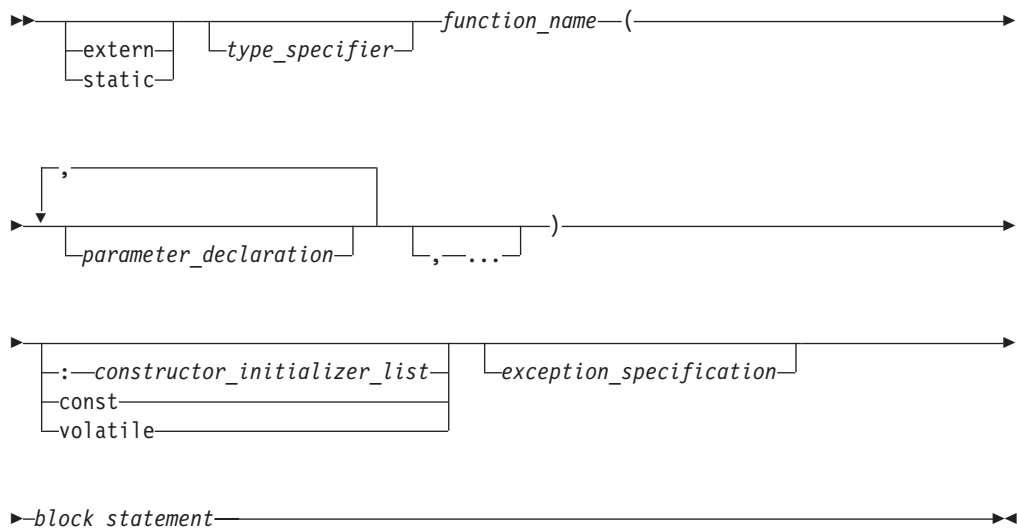
```
void f5() throw();
```

RELATED REFERENCES

- “Default Arguments in C++ Functions” on page 169
- “const and volatile Member Functions” on page 296
- “Exception Specifications” on page 412
- “extern Storage Class Specifier” on page 37
- “Chapter 4. Declarators” on page 73

Function Definitions

A *function definition* contains a function declaration and the body of a function.



A function definition contains the following:

- An optional *storage class specifier* **extern** or **static**, which determines the scope of the function. If a storage class specifier is not given, the function has external linkage.

•

A *type specifier*, which determines the type of value that the function returns.

► **C** In C, the type specifier is optional. If a type specifier is not given, the function has type **int**.

► **C++** The type specifier is not optional in C++.

•

A *function declarator*, which is the function name followed by a parenthesized list of parameter types and names. It can further describe the type of the value that

the function returns, and lists the type and name of each parameter that the function expects. In the following function definition, `f(int a, int b)` is the function declarator:

```
int f(int a, int b) {
    return a + b;
}
```

- **C++** Optional **const** or **volatile** specifiers after the function declarator. Only member functions may have these.
- **C++** An optional *exception specification*, which limits the function from throwing only a specified list of exceptions.
-

A *block statement*, which contains data definitions and code.

► **C++** You can also have a function try block instead of a block statement. If the function definition is a constructor, you can have a *constructor initializer list* before the block statement. In the following class definition, `x(0), y('c')` is a constructor initializer list:

```
class A {
    int x;
    char y;
public:
    A() : x(0), y('c') { }
};
```

A function can be called by itself or by other functions. By default, function definitions have external linkage, and can be called by functions defined in other files. A storage class specifier of **static** means that the function name has global scope only, and can be directly invoked only from within the same translation unit.

► **C++** This use of **static** is deprecated in C++. Instead, place the function in the unnamed namespace.

► **C** In C only, if a function definition has external linkage and a return type of **int**, calls to the function can be made before it is visible because an implicit declaration of `extern int func();` is assumed.

All declarations for a given function must be compatible; that is, the return type is the same and the parameters have the same type. Note that overloaded functions have the same name.

► **C** The default type for the return value and parameters of a function is **int**, and the default storage class specifier is **extern**. If the function does not return a value, use the keyword **void** as the type specifier. You can use the keyword **void** as a parameter declaration to indicate the function is not passed any arguments.

A function cannot return a function, array, or object with a **volatile** or **const** type, but it can return a pointer to these or any other types.

► **C** In C, you cannot declare a function as a struct or union member.

► **C** In C, a function cannot return any type having the **volatile** or **const** qualifier.

Function Definitions

You cannot define an array of functions. You can, however, define an array of pointers to functions.

The following example is a definition of the function sum:

```
int sum(int x,int y)
{
    return(x + y);
}
```

The function sum has external linkage, returns an object that has type **int**, and has two parameters of type **int** declared as x and y. The function body contains a single statement that returns the sum of x and y.

In the following example, ary is an array of two function pointers. Type casting is performed to the values assigned to ary for compatibility:

```
#include <stdio.h>

typedef void (*ARYTYPE)();

int func1(void);
void func2(double a);

int main(void)
{
    double num = 333.3333;
    int retnum;
    ARYTYPE ary[2];
    ary[0]=(ARYTYPE)func1;
    ary[1]=(ARYTYPE)func2;

    retnum=((int (*)( ))ary[0])();    /* calls func1 */
    printf("number returned = %i\n", retnum);
    ((void (*)(double))ary[1])(num); /* calls func2 */

    return(0);
}

int func1(void)
{
    int number=3;
    return number;
}

void func2(double a)
{
    printf("result of func2 = %f\n", a);
}
```

The following is the output of the above example:

```
number
returned = 3
result of func2 = 333.333300
```

RELATED REFERENCES

- “extern Storage Class Specifier” on page 37
- “static Storage Class Specifier” on page 42
- “Block Statement” on page 179
- “Pointers” on page 81
- “References” on page 92
- “Structures” on page 51
- “Unions” on page 59

- “volatile and const Qualifiers” on page 69

Ellipsis and void

An ellipsis at the end of the parameter specifications is used to specify that a function has a variable number of parameters. The number of parameters is equal to, or greater than, the number of parameter specifications. At least one parameter declaration must come before the ellipsis.

```
int f(int, ...);
```

► **C++** The comma before the ellipsis is optional. In addition, a parameter declaration is not required before the ellipsis.

► **C** The comma before the ellipsis as well as a parameter declaration before the ellipsis are both required in C.

Parameter promotions are performed as needed, but no type checking is done on the variable arguments.

You can declare a function with no arguments in two ways:

```
int f(void);
int f();
```

► **C++** An empty argument declaration list or the argument declaration list of (void) indicates a function that takes no arguments.

► **C** An empty argument declaration list means that the function may take any number or type of parameters.

The type **void** cannot be used as an argument type, although types derived from **void** (such as pointers to **void**) can be used.

In the following example, the function `f()` takes one integer argument and returns no value, while `g()` expects no arguments and returns an integer.

```
void f(int);
int g(void);
```

RELATED REFERENCES

- “void Type” on page 50

Examples of Function Definitions

The following example contains a function declarator `i_sort` with **table** declared as a pointer to `int` and `length` declared as type **int**. Note that arrays as parameters are implicitly converted to a pointer to the element type.

CCNRAAU

```
/**
 * This example illustrates function definitions.
 * Note that arrays as parameters are implicitly
 * converted to a pointer to the type.
 */
```

```
#include <stdio.h>
```

```
void i_sort(int table[ ], int length);
```

Function Definitions

```
int main(void)
{
    int table[ ]={1,5,8,4};
    int length=4;
    printf("length is %d\n",length);
    i_sort(table,length);
}

void i_sort(int table[ ], int length)
{
    int i, j, temp;

    for (i = 0; i < length -1; i++)
        for (j = i + 1; j < length; j++)
            if (table[i] > table[j])
            {
                temp = table[i];
                table[i] = table[j];
                table[j] = temp;
            }
}
```

The following are examples of function declarations (also called *function prototypes*):

```
double square(float x);
int area(int x,int y);
static char *search(char);
```

The following example illustrates how a **typedef** identifier can be used in a function declarator:

```
typedef struct tm_fmt { int minutes;
                        int hours;
                        char am_pm;
                    } struct_t;
long time_seconds(struct_t arrival)
```

The following function `set_date` declares a pointer to a structure of type `date` as a parameter. `date_ptr` has the storage class specifier **register**.



```
void set_date(register struct date *date_ptr)
{
    date_ptr->mon = 12;
    date_ptr->day = 25;
    date_ptr->year = 87;
}
```

RELATED REFERENCES

- “Block Statement” on page 179
- “Function Definitions” on page 158
- “Function Declarations” on page 154

The main() Function

When a program begins running, the system calls the function **main**, which marks the entry point of the program. Every program must have one function named **main**. No other function in the program can be called **main**. A **main** function has one of two forms:

-  **int main (void)** *block_statement*
-  **int main ()** *block_statement*

- **int main (int argc , char ** argv) *block_statement***

The argument *argc* is the number of command-line arguments passed to the program. The argument *argv* is a pointer to an array of strings, where *argv[0]* is the name you used to run your program from the command-line, *argv[1]* the first argument that you passed to your program, *argv[2]* the second argument, and so on.

By default, **main** has the storage class **extern**.

> C++ You cannot declare **main** as **inline** or **static**. You cannot call **main** from within a program or take the address of **main**. You cannot overload this function.

RELATED REFERENCES

- “extern Storage Class Specifier” on page 37
- “Inline Functions” on page 174
- “static Storage Class Specifier” on page 42

Arguments to main

The function **main** can be declared with or without parameters.

```
int main(int argc, char *argv[])
```

Although any name can be given to these parameters, they are usually referred to as *argc* and *argv*.

The first parameter, *argc* (argument count), has type **int** and indicates how many arguments were entered on the command line.

The second parameter, *argv* (argument vector), has type array of pointers to **char** array objects. **char** array objects are null-terminated strings.

The value of *argc* indicates the number of pointers in the array *argv*. If a program name is available, the first element in *argv* points to a character array that contains the program name or the invocation name of the program that is being run. If the name cannot be determined, the first element in *argv* points to a null character.

This name is counted as one of the arguments to the function **main**. For example, if only the program name is entered on the command line, *argc* has a value of 1 and *argv[0]* points to the program name.

Regardless of the number of arguments entered on the command line, *argv[argc]* always contains NULL.

RELATED REFERENCES

- “Integer Variables” on page 49
- “char and wchar_t Type Specifiers” on page 45

Example of Arguments to main

The following program backward prints the arguments entered on a command line such that the last argument is printed first:

main

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s ", argv[argc]);
}
```

Invoking this program from a command line with the following:

```
backward string1 string2
```

gives the following output:

```
string2 string1
```

The arguments *argc* and *argv* would contain the following values:

Object	Value
argc	3
argv[0]	pointer to string "backward"
argv[1]	pointer to string "string1"
argv[2]	pointer to string "string2"
argv[3]	NULL

Note: Be careful when entering mixed case characters on a command line because some environments are not case sensitive. Also, the exact format of the string pointed to by *argv[0]* is system dependent.

RELATED REFERENCES

- “Calling Functions and Passing Arguments”
- “Type Specifiers” on page 44
- “Identifiers” on page 18
- “Block Statement” on page 179

Calling Functions and Passing Arguments

The arguments of a function call are used to initialize the parameters of the function definition.

Integral and floating-point promotions will first be done to the values of the arguments before the function is called.

The type of an argument is checked against the type of the corresponding parameter in the function declaration. All standard and user-defined type conversions are applied as necessary.

For example:

```
#include <stdio.h>
#include <math.h>

/* Declaration */
extern double root(double, double);

/* Definition */
double root(double value, double base) {
    double temp = exp(log(value)/base);
    return temp;
}
```

Calling Functions and Passing Arguments

```
int main(void) {
    int value = 144;
    int base = 2;
    printf("The root is: %f\n", root(value, base));
    return 0;
}
```

The output is The root is: 12.000000

In the above example, because the function `root` is expecting arguments of type **double**, the two **int** arguments `value` and `base` are implicitly converted to type **double** when the function is called.

The order in which arguments are evaluated and passed to the function is implementation-defined. For example, the following sequence of statements calls the function `tester`:

```
int x;
x = 1;
tester(x++, x);
```

The call to `tester` in the example may produce different results on different compilers. Depending on the implementation, `x++` may be evaluated first or `x` may be evaluated first. To avoid the ambiguity and have `x++` evaluated first, replace the preceding sequence of statements with the following:

```
int x, y;
x = 1;
y = x++;
tester(y, x);
```

► **C++** In C++, if a nonstatic class member function is passed as an argument, the argument is converted to a pointer to member.

► **C++** If a class has a destructor or a copy constructor that does more than a bitwise copy, passing a class object by value results in the construction of a temporary that is actually passed by reference.

► **C++** It is an error when a function argument is a class object and all of the following properties hold:

- The class needs a copy constructor.
- The class does not have a user-defined copy constructor.
- A copy constructor cannot be generated for that class.

► **z/OS** You cannot pass a packed structure argument to a function that expects a nonpacked structure of the same type and vice versa. (The same applies to packed and nonpacked unions.)

RELATED REFERENCES

- “Function Calls ()” on page 104
- “Integral and Floating-Point Promotions” on page 143
- “Constructors” on page 342

Command-Line Arguments (z/OS)

► **z/OS** The maximum allowable length of a command-line argument for z/OS Language Environment is 64K.

Command-Line Arguments

z/OS C/C++ treats arguments that you enter on the command line differently in different environments. The following lists how `argv` and `argc` are handled.

Under z/OS Batch

<code>argc</code>	Returns the number of strings in the argument line
<code>argv[0]</code>	Returns the program name in uppercase
<code>argv[1 to n]</code>	Returns the arguments as you enter them

Under IMS

<code>argc</code>	Returns 1
<code>argv[0]</code>	Is a null pointer

Under CICS

<code>argc</code>	Returns 1
<code>argv[0]</code>	Returns the transaction ID

Under TSO Command

<code>argc</code>	Returns the number of strings in the argument line
<code>argv[0]</code>	Returns the program name in uppercase
<code>argv[1 to n]</code>	Returns the arguments exactly as you enter them

Under TSO Call

Without the ASIS option:

<code>argc</code>	Returns the number of strings in the argument line
<code>argv</code>	Returns the program name and arguments in lowercase

With the ASIS option:

<code>argc</code>	Returns the number of strings in the argument line
<code>argv[0]</code>	Returns the program name in uppercase
<code>argv[1 to n]</code>	Arguments entered in uppercase are returned in lowercase. Arguments entered in mixed or lowercase are returned as entered.

Under z/OS UNIX Shell

<code>argc</code>	Returns the number of strings in the argument line
<code>argv[0]</code>	Returns the program name as you enter it
<code>argv[1 to n]</code>	Returns the arguments exactly as you enter them

The only delimiter for the arguments that are passed to `main()` is white space. z/OS C/C++ uses commas passed to `main()` by JCL as arguments and not as delimiters.

The following example appends the comma to the 'one' when passed to `main()`.

```
//FUNC EXEC PCGO,GPGM='FUNC',  
//      PARM.GO=('one',  
//      'two')
```

For more information on restrictions of the command-line arguments, refer to *z/OS C/C++ User's Guide*.

RELATED REFERENCES

- “Calling Functions and Passing Arguments” on page 164
- “Type Specifiers” on page 44
- “Identifiers” on page 18
- “Block Statement” on page 179

Passing Arguments by Value

If you call a function with an argument that corresponds to a non-reference parameter, you have passed that argument by value. The parameter is initialized with the value of the argument. You can change the value of the parameter (if that parameter has not been declared **const**) within the scope of the function, but these changes will not affect the value of the argument in the calling function.

The following are examples of passing arguments by value:

The following statement calls the function **printf**, which receives a character string and the return value of the function **sum**, which receives the values of **a** and **b**:

```
printf("sum = %d\n", sum(a,b));
```

The following program passes the value of **count** to the function **increment**, which increases the value of the parameter **x** by 1.

CCNRAAX

```
/**
 ** An example of passing an argument to a function
 **/

#include <stdio.h>

void increment(int);

int main(void)
{
    int count = 5;

    /* value of count is passed to the function */
    increment(count);
    printf("count = %d\n", count);

    return(0);
}

void increment(int x)
{
    ++x;
    printf("x = %d\n", x);
}
```

The output illustrates that the value of **count** in **main** remains unchanged:

```
x = 6
count = 5
```

RELATED REFERENCES

- “Function Calls ()” on page 104

Passing Arguments by Reference

Passing by reference refers to a method of passing arguments where the value of an argument in the calling function can be modified in the called function.

Command-Line Arguments

To pass an argument by reference, you declare the corresponding parameter with a reference type.

The following example shows how arguments are passed by reference. Note that reference parameters are initialized with the actual arguments when the function is called.

CCNX06A

```
#include <stdio.h>

void swapnum(int &i, int &j) {
    int temp = i;
    i = j;
    j = temp;
}

int main(void) {
    int a = 10;
    int b = 20;

    swapnum(a, b);
    printf("A is %d and B is %d\n", a, b);
    return 0;
}
```

When the function `swapnum()` is called, the actual values of the variables `a` and `b` are exchanged because they are passed by reference. The output is:

A is 20 and B is 10

You must define the parameters of `swapnum()` as references if you want the values of the actual arguments to be modified by the function `swapnum()`.

C++ In order to modify a reference that is **const**-qualified, you must cast away its constness with the **const_cast** operator. The following example demonstrates this:

```
#include <iostream>
using namespace std;

void f(const int& x) {
    int* y = const_cast<int>(&x);
    (*y)++;
}

int main() {
    int a = 5;
    f(a);
    cout << a << endl;
}
```

This example outputs 6.

You can modify the values of nonconstant objects through pointer parameters. The following example demonstrates this:

CCNRAAY

```
#include <stdio.h>

int main(void)
{
    void increment(int *x);
}
```

```
int count = 5;

/* address of count is passed to the function */
increment(&count);
printf("count = %d\n", count);

return(0);
}

void increment(int *x)
{
    ++*x;
    printf("*x = %d\n", *x);
}
```

The following is the output of the above code:

```
*x = 6
count = 6
```

The example passes the address of count to increment(). Function increment() increments count through the pointer parameter x.

RELATED REFERENCES

- “References” on page 92
- “const_cast Operator” on page 110

Default Arguments in C++ Functions

C++ You can provide default values for function parameters. For example:

CCNX06B

```
#include <iostream>
using namespace std;

int a = 1;
int f(int a) { return a; }
int g(int x = f(a)) { return x; }

int h() {
    a = 2;
    {
        int a = 3;
        return g();
    }
}

int main() {
    cout << h() << endl;
}
```

This example prints 2 to standard output, because the a referred to in the declaration of g() is the one at file scope, which has the value 2 when g() is called.

The default argument must be implicitly convertible to the parameter type.

A pointer to a function must have the same type as the function. Attempts to take the address of a function by reference without specifying the type of the function will produce an error. The type of a function is not affected by arguments with default values.

Default Arguments in C++ Functions

The following example shows that default arguments are not considered part of a function's type. The default argument allows you to call a function without specifying all of the arguments, it does not allow you to create a pointer to the function that does not specify the types of all the arguments. Function `f` can be called without an explicit argument, but the pointer `badpointer` cannot be defined without specifying the type of the argument:

```
int f(int = 0);
void g()
{
    int a = f(1);           // ok
    int b = f();            // ok, default argument used
}
int (*pointer)(int) = &f;   // ok, type of f() specified (int)
int (*badpointer)() = &f;   // error, badpointer and f have
                           // different types. badpointer must
                           // be initialized with a pointer to
                           // a function taking no arguments.
```

RELATED REFERENCES

- “Pointers to Functions” on page 173

Restrictions on Default Arguments

Of the operators, only the function call operator and the operator **new** can have default arguments when they are overloaded.

Parameters with default arguments must be the trailing parameters in the function declaration parameter list. For example:

```
void f(int a, int b = 2, int c = 3); // trailing defaults
void g(int a = 1, int b = 2, int c); // error, leading defaults
void h(int a, int b = 3, int c);    // error, default in middle
```

Once a default argument has been given in a declaration or definition, you cannot redefine that argument, even to the same value. However, you can add default arguments not given in previous declarations. For example, the last declaration below attempts to redefine the default values for `a` and `b`:

```
void f(int a, int b, int c=1); // valid
void f(int a, int b=1, int c); // valid, add another default
void f(int a=1, int b, int c); // valid, add another default
void f(int a=1, int b=1, int c=1); // error, redefined defaults
```

You can supply any default argument values in the function declaration or in the definition. Any parameters in the parameter list following a default argument value must have a default argument value specified in this or a previous declaration of the function.

You cannot use local variables in default argument expressions. For example, the compiler generates errors for both function `g()` and function `h()` below:

```
void f(int a)
{
    int b=4;
    void g(int c=a); // Local variable "a" cannot be used here
    void h(int d=b); // Local variable "b" cannot be used here
}
```

RELATED REFERENCES

- “Function Calls ()” on page 104
- “C++ new Operator” on page 119
- “Default Arguments in C++ Functions” on page 169

Evaluating Default Arguments

When a function defined with default arguments is called with trailing arguments missing, the default expressions are evaluated. For example:

```
void f(int a, int b = 2, int c = 3); // declaration
// ...
int a = 1;
f(a);           // same as call f(a,2,3)
f(a,10);        // same as call f(a,10,3)
f(a,10,20);     // no default arguments
```

Default arguments are checked against the function declaration and evaluated when the function is called. The order of evaluation of default arguments is undefined. Default argument expressions cannot use other parameters of the function. For example:

```
int f(int q = 3, int r = q); // error
```

The argument `r` cannot be initialized with the value of the argument `q` because the value of `q` may not be known when it is assigned to `r`. If the above function declaration is rewritten:

```
int q=5;
int f(int q = 3, int r = q); // error
```

The value of `r` in the function declaration still produces an error because the variable `q` defined outside of the function is hidden by the argument `q` declared for the function. Similarly:

```
typedef double D;
int f(int D, int z = D(5.3) ); // error
```

Here the type `D` is interpreted within the function declaration as the name of an integer. The type `D` is hidden by the argument `D`. The cast `D(5.3)` is therefore not interpreted as a cast because `D` is the name of the argument not a type.

In the following example, the nonstatic member `a` cannot be used as an initializer because `a` does not exist until an object of class `X` is constructed. You can use the static member `b` as an initializer because `b` is created independently of any objects of class `X`. You can declare the member `b` after its use as a default argument because the default values are not analyzed until after the final bracket `}` of the class declaration.

```
class X
{
    int a;
    f(int z = a) ; // error
    g(int z = b) ; // valid
    static int b;
};
```

RELATED REFERENCES

- “Default Arguments in C++ Functions” on page 169

Function Return Values

You must return a value from a function unless the function has a return type of `void`.

Function Return Values

C All functions must return a value, including those that have a return type of **void**. The return value is specified in a **return** statement. The following code fragment shows a function definition, including the **return** statement:

```
int add(int i, int j)
{
    return i + j; // return statement
}
```

The function `add()` can be called as shown in the following code fragment:

```
int a = 10,
    b = 20;
int answer = add(a, b); // answer is 30
```

In this example, the return statement initializes a variable of the returned type. The variable `answer` is initialized with the **int** value 30. The type of the returned expression is checked against the returned type. All standard and user-defined conversions are performed as necessary.

The following **return** statements show different ways of returning values to a caller:

```
return;                // Returns no value
return result;         // Returns the value of result
return 1;              // Returns the value 1
return (x * x);        // Returns the value of x * x
```

Each time a function is called, new copies of its variables with automatic storage are created. Because the storage for these automatic variables may be reused after the function has terminated, a pointer or reference to an automatic variable should not be returned.

C++ If a class object is returned, a temporary object may be created if the class has copy constructors or a destructor.

RELATED REFERENCES

- “auto Storage Class Specifier” on page 35
- “Temporary Objects” on page 357
- “Destructors” on page 350

Using References as Return Types

References can also be used as return types for functions. The reference returns the lvalue of the object to which it refers. This allows you to place function calls on the left side of assignment statements.

C++ Referenced return values are used when assignment operators and subscripting operators are overloaded so that the results of the overloaded operators can be used as actual values.

Note: Returning a reference to an automatic variable gives unpredictable results.

RELATED REFERENCES

- “Overloading Assignments” on page 274
- “Overloading Subscripting” on page 277
- “auto Storage Class Specifier” on page 35

Pointers to Functions

A pointer to a function points to the address of the executable code of the function. You can use pointers to call functions and to pass functions as arguments to other functions. You cannot perform pointer arithmetic on pointers to functions.

z/OS For z/OS C/C++, use the `__cdecl` keyword to declare a pointer to a function as a C linkage. For more information, refer to “`__cdecl` Keyword (z/OS C++ Only)” on page 75.

The type of a pointer to a function is based on both the return type and parameter types of the function.

A declaration of a pointer to a function must have the pointer name in parentheses. The function call operator `()` has a higher precedence than the dereference operator `*`. Without them, the compiler interprets the statement as a function that returns a pointer to a specified return type. For example:

```
int *f(int a);      // function f returning an int*
int (*g)(int a);   // pointer g to a function returning an int
```

In the first declaration, `f` is interpreted as a function that takes an **int** as argument, and returns a pointer to an **int**. In the second declaration, `g` is interpreted as a pointer to a function that takes an **int** argument and that returns an **int**.

z/OS Under z/OS C/C++, if you pass a function pointer to a function, or the function returns a function pointer, the declared or implied linkages must be the same. Use the `extern` keyword with declarations in order to specify different linkages.

The following example illustrates the correct and incorrect uses of function pointers under z/OS C/C++ :

```
#include <stdlib.h>

extern "C"    int cf();
extern "C++" int cxxf(); // C++ is included here for clarity;
                        // it is not required; if it is
                        // omitted, cxxf() will still have
                        // C++ linkage.

extern "C"    int (*c_fp)();
extern "C++"  int (*cxx_fp)();
typedef int (*dft_fp_T)();
typedef int (dft_f_T)();

extern "C" {
    typedef void (*cfp_T)();
    typedef int (*cf_pT)();
    void cfn();
    void (*cfp)();
}

extern "C++" {
    typedef int (*cxxf_pT)();
    void cxxfn();
    void (*cxxfp)();
}

extern "C" void f_cprm(int (*f)()) {
    int (*s)() = cxxf; // error, incompatible linkages-cxxf has
                      // C++ linkage, s has C linkage as it
```

Pointers to Functions

```

                                // is included in the extern "C" wrapper
cxxf_pT j = cxxf;              // valid, both have C++ linkage
int (*i)() = cf;               // valid, both have C linkage
}

extern "C++" void f_cxprm(int (*f)()) {
    int (*s)() = cf;           // error, incompatible linkages-cf has C
                                // linkage, s has C++ linkage as it is
                                // included in the extern "C++" wrapper
    int (*i)() = cxxf;         // valid, both have C++ linkage
    cf_pT j = cf;              // valid, both have C linkage
}

main() {

    c_fp = cxxf;                // error - c_fp has C linkage and cxxf has
                                // C++ linkage
    cxx_fp = cf;                // error - cxx_fp has C++ linkage and
                                // cf has C linkage
    dft_fp_T dftfpT1 = cf;      // error - dftfpT1 has C++ linkage and
                                // cf has C linkage
    dft_f_T *dftfT3 = cf;       // error - dftfT3 has C++ linkage and
                                // cf has C linkage
    dft_fp_T dftfpT5 = cxxf;     // valid
    dft_f_T *dftfT6 = cxxf;     // valid

    c_fp = cf;                  // valid
    cxx_fp = cxxf;              // valid
    f_cprm(cf);                 // valid
    f_cxprm(cxxf);              // valid

    // The following errors are due to incompatible linkage of function
    // arguments, type conversion not possible
    f_cprm(cxxf);               // error - f_cprm expects a parameter with
                                // C linkage, but cxxf has C++ linkage
    f_cxprm(cf);                // error - f_cxprm expects a parameter
                                // with C++ linkage, but cf has C linkage
}
```

For z/OS, linkage compatibility affects all C library functions that accept a function pointer as a parameter. The `qsort()` function is an example of these functions.

RELATED REFERENCES

- “Pointers” on page 81
- “Pointer Conversions” on page 146
- “extern Storage Class Specifier” on page 37

Inline Functions

C++ A function is declared inline by using the **inline** function specifier or by defining a member function within a class or structure definition.

The **inline** specifier is a suggestion to the compiler that an inline expansion can be performed. Instead of transferring control to and from the function code segment, a modified copy of the function body may be substituted directly for the function call.

An inline function can be declared and defined simultaneously. If it is declared with the keyword **inline**, it can be declared without a definition. The following code fragment shows an inline function definition. Note that the definition includes both the declaration and body of the inline function.

```
inline int add(int i, int j) { return i + j; }
```


Both member and nonmember functions can be inline, and both have external linkage by default.

The use of the **inline** specifier does not change the meaning of the function. The inline expansion of a function may not preserve the order of evaluation of the actual arguments.


RELATED REFERENCES

- “Member Functions” on page 295

Inline Functions

Chapter 8. Statements

A statement is the smallest independent computational unit. The following is a summary of the statements available in C and C++:

- labeled statements
 - identifier labels
 - **case** labels
 - **default** labels
- expression statements
- blocks or compound statements
- selection statements
 - **if** statements
 - **switch** statements
- iteration statements
 - **while** statements
 - **do** statements
 - **for** statements
- jump statements
 - **break** statements
 - **continue** statements
 - **return** statements
 - **goto** statements
- declaration statements
-  **try** blocks

Labels

There are three kinds of labels: identifier, case, and default.

Identifier label statements have the following form:

►► *identifier* : *statement* ◀◀

The label consists of the *identifier* and the colon (:) character. An identifier label is only used as the target of a **goto** statement. Identifier labels have their own namespace; you do not have to worry about identifier labels conflicting with other identifiers.

Case statements have the following form:

►► *case* *constant_expression* : *statement* ◀◀

Default label statements have the following form:

►► *default* : *statement* ◀◀

Case and default label statements only appear in **switch** statements.

Examples of Labels

Labels

```
comment_complete : ;           /* null statement label */
test_for_null : if (NULL == pointer)
```

RELATED REFERENCES

- “goto Statement” on page 193
- “switch Statement” on page 182

Expression Statements

An *expression statement* contains an expression. The expression can be null.

An expression statement has the form:

```
┌──────────┐
└expression┘ ;
```

An expression statement evaluates *expression*, then discards the value of the expression. An expression statement without an expression is a null statement.

Examples of Expressions

```
printf("Account Number: \n");    /* call to the printf */
marks = dollars * exch_rate;      /* assignment to marks */
(difference < 0) ? ++losses : ++gain; /* conditional increment */
```

RELATED REFERENCES

- “Chapter 5. Expressions and Operators” on page 95

Resolving Ambiguous Statements in C++

► C++ The C++ syntax does not disambiguate between expression statements and declaration statements. The ambiguity arises when an expression statement has a function-style cast as its left-most subexpression. (Note that, because C does not support function-style casts, this ambiguity does not occur in C programs.) If the statement can be interpreted both as a declaration and as an expression, the statement is interpreted as a declaration statement.

Note: The ambiguity is resolved only on a syntactic level. The disambiguation does not use the meaning of the names, except to assess whether or not they are type names.

The following expressions disambiguate into expression statements because the ambiguous subexpression is followed by an assignment or an operator. `type_spec` in the expressions can be any type specifier:

```
type_spec(i)++;           // expression statement
type_spec(i,3)<<d;         // expression statement
type_spec(i)->l=24;        // expression statement
```

In the following examples, the ambiguity cannot be resolved syntactically, and the statements are interpreted as declarations. `type_spec` is any type specifier:

```
type_spec(*i)(int);        // declaration
type_spec(j)[5];           // declaration
type_spec(m) = { 1, 2 };   // declaration
type_spec(*k) (float(3));  // declaration
```

The last statement above causes a compile-time error because you cannot initialize a pointer with a float value.

Any ambiguous statement that is not resolved by the above rules is by default a declaration statement. All of the following are declaration statements:

```
type_spec(a);           // declaration
type_spec(*b)();        // declaration
type_spec(c)=23;        // declaration
type_spec(d),e,f,g=0;    // declaration
type_spec(h)(e,3);       // declaration
```

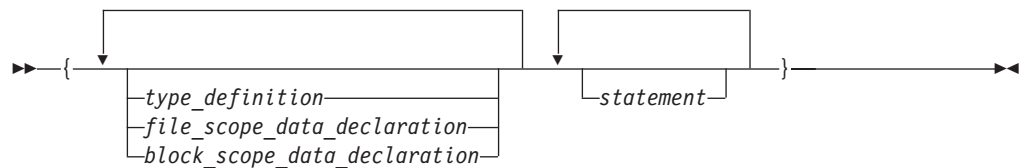
RELATED REFERENCES

- “Chapter 3. Declarations” on page 33
- “Chapter 5. Expressions and Operators” on page 95
- “Function Calls ()” on page 104

Block Statement

A *block statement*, or *compound statement*, lets you group any number of data definitions, declarations, and statements into one statement. All definitions, declarations, and statements enclosed within a single set of braces are treated as a single statement. You can use a block wherever a single statement is allowed.

A block statement has the form:



C In C, Any definitions and declarations must come before the statements.

A block defines a local scope. If a data object is usable within a block and its identifier is not redefined, all nested blocks can use that data object.

Example of Blocks

The following program shows how the values of data objects change in nested blocks:

```
/**
** This example shows how data objects change in nested blocks.
**/
#include <stdio.h>

int main(void)
{
    int x = 1;           /* Initialize x to 1 */
    int y = 3;

    if (y > 0)
    {
        int x = 2;       /* Initialize x to 2 */
        printf("second x = %4d\n", x);
    }
}
```

Block Statement

```
    printf("first  x = %4d\n", x);  
    return(0);  
}
```

The program produces the following output:

```
second x = 2  
first  x =  1
```

Two variables named `x` are defined in `main`. The first definition of `x` retains storage while `main` is running. However, because the second definition of `x` occurs within a nested block, `printf("second x = %4d\n", x);` recognizes `x` as the variable defined on the previous line. Because `printf("first x = %4d\n", x);` is not part of the nested block, `x` is recognized as the first definition of `x`.

RELATED REFERENCES

- “Storage Class Specifiers” on page 34
- “Type Specifiers” on page 44

if Statement

C++ An *if statement* lets you conditionally process a statement when the specified test expression, implicitly converted to **bool**, evaluates to **true**. If the implicit conversion to **bool** fails the program is ill-formed.

C In C, an **if** statement lets you conditionally process a statement when the specified test expression evaluates to a nonzero value. The test expression must be of arithmetic or pointer type.

You can optionally specify an **else** clause on the **if** statement. If the test expression evaluates to **false** (or in C, a zero value) and an **else** clause exists, the statement associated with the **else** clause runs. If the test expression evaluates to **true**, the statement following the expression runs and the **else** clause is ignored.

An **if** statement has the form:

```
►► if (—expression—) —statement—  
    else —statement— ◄◄
```

When **if** statements are nested and **else** clauses are present, a given **else** is associated with the closest preceding **if** statement within the same block.

A single statement following any selection statements (**if**, **switch**) is treated as a compound statement containing the original statement. As a result any variables declared on that statement will be out of scope after the **if** statement. For example:

```
if (x)  
int i;
```

is equivalent to:

```
if (x)  
{ int i; }
```

Variable `i` is visible only within the **if** statement. The same rule applies to the **else** part of the **if** statement.

Examples of if Statements

The following example causes grade to receive the value A if the value of score is greater than or equal to 90.

```
if (score >= 90)
    grade = 'A';
```

The following example displays Number is positive if the value of number is greater than or equal to 0. If the value of number is less than 0, it displays Number is negative.

```
if (number >= 0)
    printf("Number is positive\n");
else
    printf("Number is negative\n");
```

The following example shows a nested **if** statement:

```
if (paygrade == 7)
    if (level >= 0 && level <= 8)
        salary *= 1.05;
    else
        salary *= 1.04;
else
    salary *= 1.06;
cout << "salary is " << salary << endl;
```

The following example shows a nested **if** statement that does not have an **else** clause. Because an **else** clause always associates with the closest **if** statement, braces might be needed to force a particular **else** clause to associate with the correct **if** statement. In this example, omitting the braces would cause the **else** clause to associate with the nested **if** statement.

```
if (kegs > 0) {
    if (furlongs > kegs)
        fpk = furlongs/kegs;
}
else
    fpk = 0;
```

The following example shows an **if** statement nested within an **else** clause. This example tests multiple conditions. The tests are made in order of their appearance. If one test evaluates to a nonzero value, a statement runs and the entire **if** statement ends.

```
if (value > 0)
    ++increase;
else if (value == 0)
    ++break_even;
else
    ++decrease;
```

RELATED REFERENCES

- “Boolean Variables” on page 46

switch Statement

A *switch statement* lets you transfer control to different statements within the **switch** body depending on the value of the switch expression. The **switch** expression must evaluate to an integral or enumeration value. The body of the **switch** statement contains *case clauses* that consist of

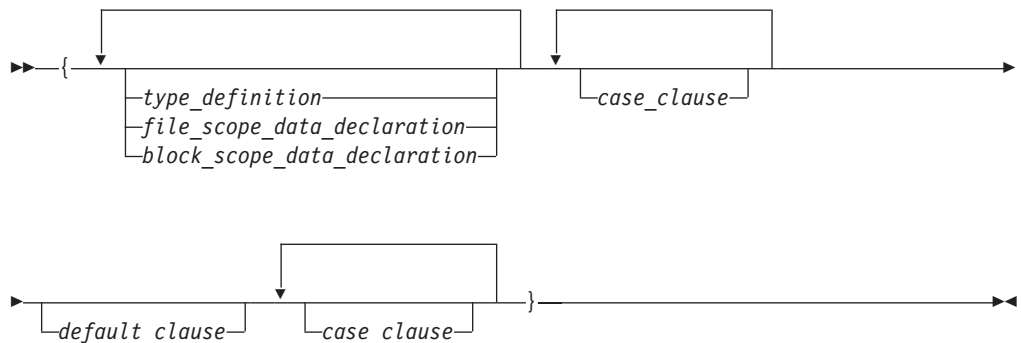
- A **case** label
- An optional **default** label
- A **case** expression
- A list of statements.

If the value of the **switch** expression equals the value of one of the case expressions, the statements following that case expression are processed. If not, the default label statements, if any, are processed.

A **switch** statement has the form:

```
switch (expression) switch_body
```

The *switch body* is enclosed in braces and can contain definitions, declarations, *case clauses*, and a *default clause*. Each case clause and default clause can contain statements.

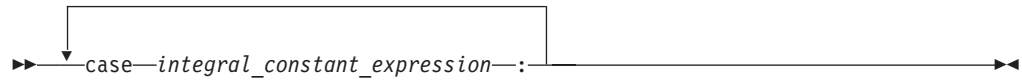


Note: An initializer within a *type_definition*, *file_scope_data_declaration* or *block_scope_data_declaration* is ignored.

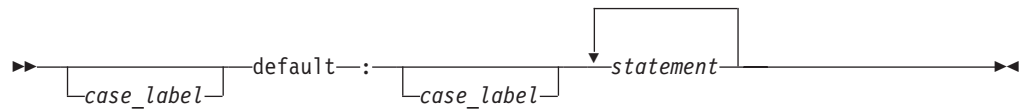
A *case clause* contains a *case label* followed by any number of statements. A case clause has the form:

```
case_label statement
```

A *case label* contains the word **case** followed by an integral constant expression and a colon. The value of each integral constant expression must represent a different value; you cannot have duplicate **case** labels. Anywhere you can put one **case** label, you can put multiple **case** labels. A case label has the form:



A *default clause* contains a **default** label followed by one or more statements. You can put a **case** label on either side of the **default** label. A **switch** statement can have only one **default** label. A *default_clause* has the form:



The **switch** statement passes control to the statement following one of the labels or to the statement following the **switch** body. The value of the expression that precedes the **switch** body determines which statement receives control. This expression is called the *switch expression*.

The value of the **switch** expression is compared with the value of the expression in each **case** label. If a matching value is found, control is passed to the statement following the **case** label that contains the matching value. If there is no matching value but there is a **default** label in the **switch** body, control passes to the **default** labelled statement. If no matching value is found, and there is no **default** label anywhere in the **switch** body, no part of the **switch** body is processed.

When control passes to a statement in the **switch** body, control only leaves the **switch** body when a **break** statement is encountered or the last statement in the **switch** body is processed.

If necessary, an integral promotion is performed on the controlling expression, and all expressions in the **case** statements are converted to the same type as the controlling expression. The **switch** expression can also be of class type if there is a single conversion to integral or enumeration type.

You can put data definitions at the beginning of the **switch** body, but the compiler does not initialize **auto** and **register** variables at the beginning of a **switch** body. You can have declarations in the body of the **switch** statement.

You cannot use a **switch** statement to jump over initializations.

➤ C++ In C++, you cannot transfer control over a declaration containing an explicit or implicit initializer unless the declaration is located in an inner block that is completely bypassed by the transfer of control. All declarations within the body of a **switch** statement that contain initializers must be contained in an inner block.

Examples of switch Statements

The following **switch** statement contains several **case** clauses and one **default** clause. Each clause contains a function call and a **break** statement. The **break** statements prevent control from passing down through each statement in the **switch** body.

switch Statement

If the **switch** expression evaluated to '/', the switch statement would call the function divide. Control would then pass to the statement following the **switch** body.

```
char key;

printf("Enter an arithmetic operator\n");
scanf("%c",&key);

switch (key)
{
    case '+':
        add();
        break;

    case '-':
        subtract();
        break;

    case '*':
        multiply();
        break;

    case '/':
        divide();
        break;

    default:
        printf("invalid key\n");
        break;
}
```

If the switch expression matches a case expression, the statements following the case expression are processed until a **break** statement is encountered or the end of the **switch** body is reached. In the following example, **break** statements are not present. If the value of text[i] is equal to 'A', all three counters are incremented. If the value of text[i] is equal to 'a', lettera and total are increased. Only total is increased if text[i] is not equal to 'A' or 'a'.

```
char text[100];
int capa, lettera, total;

// ...

for (i=0; i<sizeof(text); i++) {

    switch (text[i])
    {
        case 'A':
            capa++;
        case 'a':
            lettera++;
        default:
            total++;
    }
}
```

The following **switch** statement performs the same statements for more than one **case** label:

CCNRAB1

```
/**
** This example contains a switch statement that performs
** the same statement for more than one case label.
**/
```

```

#include <stdio.h>

int main(void)
{
    int month;

    /* Read in a month value */
    printf("Enter month: ");
    scanf("%d", &month);

    /* Tell what season it falls into */
    switch (month)
    {
        case 12:
        case 1:
        case 2:
            printf("month %d is a winter month\n", month);
            break;

        case 3:
        case 4:
        case 5:
            printf("month %d is a spring month\n", month);
            break;

        case 6:
        case 7:
        case 8:
            printf("month %d is a summer month\n", month);
            break;

        case 9:
        case 10:
        case 11:
            printf("month %d is a fall month\n", month);
            break;

        case 66:
        case 99:
        default:
            printf("month %d is not a valid month\n", month);
    }

    return(0);
}

```

If the expression month has the value 3, control passes to the statement:

```

printf("month %d is a spring month\n",
month);

```

The **break** statement passes control to the statement following the **switch** body.

RELATED REFERENCES

- "break Statement" on page 190

while Statement

A *while statement* repeatedly runs the body of a loop until the controlling expression evaluates to **false** (or 0 in C).

A **while** statement has the form:

```
▶ while (—expression—) —statement— ▶
```

The expression is evaluated to determine whether or not to process the body of the loop.

▶ C++ The expression must be convertible to **bool**.

▶ C The *expression* must be of arithmetic or pointer type.

If the expression evaluates to **false**, the body of the loop never runs. If the expression does not evaluate to **false**, the loop body is processed. After the body has run, control passes back to the expression. Further processing depends on the value of the condition.

A **break**, **return**, or **goto** statement can cause a **while** statement to end, even when the condition does not evaluate to **false**.

Example of while Statements

In the following program, `item[index]` triples and is printed out, as long as the value of the expression `++index` is less than `MAX_INDEX`. When `++index` evaluates to `MAX_INDEX`, the **while** statement ends.

CCNRAA7

```
/**
 ** This example illustrates the while statement.
 **/

#define MAX_INDEX (sizeof(item) / sizeof(item[0]))
#include <stdio.h>

int main(void)
{
    static int item[ ] = { 12, 55, 62, 85, 102 };
    int index = 0;

    while (index < MAX_INDEX)
    {
        item[index] *= 3;
        printf("item[%d] = %d\n", index, item[index]);
        ++index;
    }

    return(0);
}
```

RELATED REFERENCES

- “break Statement” on page 190
- “do Statement” on page 187

- “for Statement” on page 188
- “continue Statement” on page 190
- “return Statement” on page 192
- “goto Statement” on page 193
- “Boolean Variables” on page 46

do Statement

A *do statement* repeatedly runs a statement until the test expression evaluates to **false** (or 0 in C). Because of the order of processing, the statement is run at least once.

A **do** statement has the form:

► `do—statement—while—(—expression—);` ►

► **C++** The controlling *expression* must be convertible to type **bool**.

► **C** The *expression* must be of arithmetic or pointer type.

The body of the loop is run before the controlling **while** clause is evaluated. Further processing of the **do** statement depends on the value of the **while** clause. If the **while** clause does not evaluate to **false**, the statement runs again. When the **while** clause evaluates to **false**, the statement ends.

A **break**, **return**, or **goto** statement can cause the processing of a **do** statement to end, even when the **while** clause does not evaluate to **false**.

Example of do Statements

The following example keeps incrementing *i* while *i* is less than 5:

```
#include <stdio.h>

int main(void) {
    int i = 0;
    do {
        i++;
        printf("Value of i: %d\n", i);
    }
    while (i < 5);
    return 0;
}
```

The following is the output of the above example:

```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

RELATED REFERENCES

- “break Statement” on page 190
- “return Statement” on page 192
- “goto Statement” on page 193
- “Boolean Variables” on page 46

- Evaluate an expression before the first iteration of the statement (*initialization*)
- Specify an expression to determine whether or not the statement should be processed (the *condition*)
- Evaluate an expression after each iteration of the statement (often used to increment for each iteration)
- Repeatedly process the statement if the controlling part does not evaluate to **false** (or 0 in C).

▶ `for` (`expression1` ; `expression2` ; `expression3`)
 ▶ `statement`

Expression2 Is the *conditional expression*. It is evaluated before each iteration of the *statement*.

If it evaluates to **false** (or 0 in C), the statement is not processed and control moves to the next statement following the **for** statement. If *expression2* does not evaluate to **false**, the statement is processed. If you omit *expression2*, it is as if the expression had been replaced by **true**, and the **for** statement is not terminated by failure of this condition.

A **break**, **return**, or **goto** statement can cause a **for** statement to end, even when the second expression does not evaluate to false. If you omit *expression2*, you must use a **break**, **return**, or **goto** statement to end the **for** statement.

VAC++ You can set a compile option where a variable declared in the scope of a **for** statement is not local to the **for** statement.

188 C/C++ Language Reference

The following **for** statement prints the value of count 20 times. The **for** statement initially sets the value of count to 1. After each iteration of the statement, count is incremented.

```
int count;
for (count = 1; count <= 20; count++)
    printf("count = %d\n", count);
```

The following sequence of statements accomplishes the same task. Note the use of the **while** statement instead of the **for** statement.

```
int count = 1;
while (count <= 20)
{
    printf("count = %d\n", count);
    count++;
}
```

The following **for** statement does not contain an initialization expression:

```
for (; index > 10; --index)
{
    list[index] = var1 + var2;
    printf("list[%d] = %d\n", index,
        list[index]);
}
```

The following **for** statement will continue running until scanf receives the letter e:

```
for (;;)
{
    scanf("%c", &letter);
    if (letter == '\n')
        continue;
    if (letter == 'e')
        break;
    printf("You entered the letter %c\n", letter);
}
```

The following **for** statement contains multiple initializations and increments. The comma operator makes this construction possible. The first comma in the **for** expression is a punctuator for a declaration. It declares and initializes two integers, i and j. The second comma, a comma operator, allows both i and j to be incremented at each step through the loop.

```
for (int i = 0,
    j = 50; i < 10; ++i, j += 50)
{
    cout << "i = " << i << "and j = " << j
        << endl;
}
```

The following example shows a nested **for** statement. It prints the values of an array having the dimensions [5][3].

```
for (row = 0; row < 5; row++)
    for (column = 0; column < 3; column++)
        printf("%d\n",
            table[row][column]);
```

The outer statement is processed as long as the value of row is less than 5. Each time the outer **for** statement is executed, the inner **for** statement sets the initial value of column to zero and the statement of the inner **for** statement is executed 3 times. The inner statement is executed as long as the value of column is less than 3.

for Statement

RELATED REFERENCES

- “break Statement”
- “return Statement” on page 192
- “goto Statement” on page 193
- “Boolean Variables” on page 46

break Statement

A *break statement* lets you end an *iterative* (**do**, **for**, or **while**) statement or a **switch** statement and exit from it at any point other than the logical end. A **break** may only appear on one of these statements.

A **break** statement has the form:

►►—break—;—►►

In an iterative statement, the **break** statement ends the loop and moves control to the next statement outside the loop. Within nested statements, the **break** statement ends only the smallest enclosing **do**, **for**, **switch**, or **while** statement.

In a **switch** statement, the **break** passes control out of the **switch** body to the next statement outside the **switch** statement.

RELATED REFERENCES

- “do Statement” on page 187
- “for Statement” on page 188
- “while Statement” on page 186
- “switch Statement” on page 182

continue Statement

A *continue statement* ends the current iteration of a loop. Program control is passed from the **continue** statement to the end of the loop body.

A **continue** statement has the form:

►►—continue—;—►►

A **continue** statement can only appear within the body of an iterative statement.

The **continue** statement ends the processing of the action part of an iterative (**do**, **for**, or **while**) statement and moves control to the loop continuation portion of the statement. For example, if the iterative statement is a **for** statement, control moves to the third expression in the condition part of the statement, then to the second expression (the test) in the condition part of the statement.

Within nested statements, the **continue** statement ends only the current iteration of the **do**, **for**, or **while** statement immediately enclosing it.

Examples of continue Statements

The following example shows a **continue** statement in a **for** statement. The **continue** statement causes processing to skip over those elements of the array rates that have values less than or equal to 1.

CCNRAA3

```
/**
 ** This example shows a continue statement in a for statement.
 **/

#include <stdio.h>
#define SIZE 5

int main(void)
{
    int i;
    static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

    printf("Rates over 1.00\n");
    for (i = 0; i < SIZE; i++)
    {
        if (rates[i] <= 1.00) /* skip rates <= 1.00 */
            continue;
        printf("rate = %.2f\n", rates[i]);
    }

    return(0);
}
```

The program produces the following output:

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

The following example shows a **continue** statement in a nested loop. When the inner loop encounters a number in the array strings, that iteration of the loop ends. Processing continues with the third expression of the inner loop. The inner loop ends when the `'\0'` escape sequence is encountered.

CCNRAA4

```
/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters. The count excludes
 ** the digits 0 through 9.
 **/

#include <stdio.h>
#define SIZE 3

int main(void)
{
    static char *strings[SIZE] = { "ab", "c5d", "e5" };
    int i;
    int letter_count = 0;
    char *pointer;
    for (i = 0; i < SIZE; i++)
        /* for each string */
        /* for each character */
        for (pointer = strings[i]; *pointer != '\0';
            ++pointer)
        {
            /* if a number */
            if (*pointer >= '0' && *pointer <= '9')
                continue;
            letter_count++;
        }
}
```

continue Statement

```
    }  
    printf("letter count = %d\n", letter_count);  
    return(0);  
}
```

The program produces the following output:

```
letter count = 5
```

RELATED REFERENCES

- “do Statement” on page 187
- “for Statement” on page 188
- “while Statement” on page 186

return Statement

A *return statement* ends the processing of the current function and returns control to the caller of the function.

A **return** statement has the form:

►—return—expression—;—————►

A **return** statement in a function is optional. The compiler issues a warning if a return statement is not found in a function declared with a return type. If the end of a function is reached without encountering a **return** statement, control is passed to the caller as if a **return** statement without an expression were encountered. A function can contain multiple **return** statements.

RELATED REFERENCES

- “Chapter 7. Functions” on page 153

Value of a return Expression and Function Value

If an expression is present on a **return** statement, the value of the expression is returned to the caller. If the data type of the expression is different from the function return type, conversion of the return value takes place as if the value of the expression were assigned to an object with the same function return type.

If an expression is not present on a **return** statement, the value of the **return** statement is undefined. If an expression is not given on a **return** statement in a function declared with a nonvoid return type, an error message is issued, and the result of calling the function is unpredictable. For example:

```
int func1()  
{  
    return;  
}  
int func2()  
{  
    return (4321);  
}  
int main() {  
    int a=func1(); // result is unpredictable!  
    int b=func2();  
}
```

You cannot use a **return** statement with an expression when the function is declared as returning type **void**.

Examples of return Statements

```
return;           /* Returns no value          */
return result;    /* Returns the value of result */
return 1;         /* Returns the value 1        */
return (x * x);   /* Returns the value of x * x */
```

The following function searches through an array of integers to determine if a match exists for the variable `number`. If a match exists, the function `match` returns the value of `i`. If a match does not exist, the function `match` returns the value `-1` (negative one).

```
int match(int number, int array[ ], int n)
{
    int i;

    for (i = 0; i < n; i++)
        if (number == array[i])
            return (i);
    return(-1);
}
```

goto Statement

A *goto statement* causes your program to unconditionally transfer control to the statement associated with the label specified on the **goto** statement.

A **goto** statement has the form:

►► `goto label_identifier;` ◄◄

Because the **goto** statement can interfere with the normal sequence of processing, it makes a program more difficult to read and maintain. Often, a **break** statement, a **continue** statement, or a function call can eliminate the need for a **goto** statement.

You cannot use a **goto** statement to jump over initializations.

If an active block is exited using a **goto** statement, any local variables are destroyed when control is transferred from that block.

Example of goto Statements

The following example shows a **goto** statement that is used to jump out of a nested loop. This function could be written without using a **goto** statement.

CCNRAA6

```
/**
 ** This example shows a goto statement that is used to
 ** jump out of a nested loop.
 **/

#include <stdio.h>
void display(int matrix[3][3]);

int main(void)
{
```

goto Statement

```
int matrix[3][3]=    {1,2,3,4,5,2,8,9,10};
display(matrix);
return(0);
}

void display(int matrix[3][3])
{
    int i, j;

    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
        {
            if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
                goto out_of_bounds;
            printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
        }
    return;
    out_of_bounds: printf("number must be 1 through 6\n");
}
```

RELATED REFERENCES

- “break Statement” on page 190
- “continue Statement” on page 190
- “Labels” on page 177

Null Statement

The *null statement* performs no operation. It has the form:

▶▶;—————▶▶

A null statement can hold the label of a labeled statement or complete the syntax of an iterative statement.

Examples of Null Statements

The following example initializes the elements of the array price. Because the initializations occur within the **for** expressions, a statement is only needed to finish the **for** syntax; no operations are required.

```
for (i = 0; i < 3; price[i++] = 0)
    ;
```

A null statement can be used when a label is needed before the end of a block statement. For example:

```
void func(void) {
    if (error_detected)
        goto depart;
    /* further processing */
    depart: ; /* null statement required */
}
```

Chapter 9. Preprocessor Directives

The preprocessor is a program that is invoked by the compiler to process code before compilation. Directives for that program are discussed in this section. After an overview of preprocessor directives, the topics covered include textual macros, file inclusion, ISO standard and z/OS predefined macro names, conditional compilation directives, and pragmas.

RELATED REFERENCES

- “Preprocessor Overview”
- “z/OS C/C++ Predefined Macro Names” on page 205
- “z/OS Pragma Directives” on page 220

Preprocessor Overview

Preprocessing is a preliminary operation on C and C++ files before they are passed to the compiler. It allows you to do the following:

- Replace tokens in the current file with specified replacement tokens
- Imbed files within the current file
- Conditionally compile sections of the current file
- Generate diagnostic messages
- Change the line number of the next line of source and change the file name of the current file
- Apply machine-specific rules to specified sections of code

A *token* is a series of characters delimited by white space. The only white space allowed on a preprocessor directive is the space, horizontal tab, vertical tab, form feed, and comments. The new-line character can also separate preprocessor tokens.

The preprocessed source program file must be a valid C or C++ program.

The preprocessor is controlled by the following directives:

#define	Defines a macro.
#undef	Removes a preprocessor macro definition.
#error	Defines text for a compile-time error message.
#include	Inserts text from another source file.
#if	Conditionally suppresses portions of source code, depending on the result of a constant expression.
#ifdef	Conditionally includes source text if a macro name is defined.
#ifndef	Conditionally includes source text if a macro name is not defined.
#else	Conditionally includes source text if the previous #if , #ifdef , #ifndef , or #elif test fails.
#elif	Conditionally includes source text if the previous #if , #ifdef , #ifndef , or #elif test fails, depending on the value of a constant expression.
#endif	Ends conditional text.
#line	Supplies a line number for compiler messages.

Preprocessor Overview

#pragma Specifies implementation-defined instructions to the compiler.

RELATED REFERENCES

- “Tokens” on page 11
- “Preprocessor Directive Format”

Preprocessor Directive Format

Preprocessor directives begin with the # token followed by a preprocessor keyword. The # token must appear as the first character that is not white space on a line. The # is not part of the directive name and can be separated from the name with white spaces.

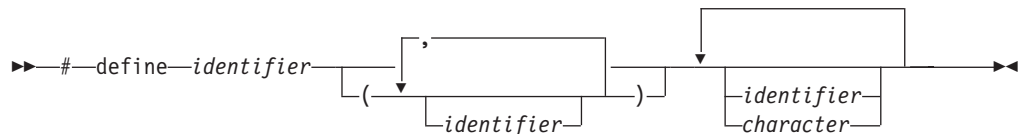
A preprocessor directive ends at the new-line character unless the last character of the line is the \ (backslash) character. If the \ character appears as the last character in the preprocessor line, the preprocessor interprets the \ and the new-line character as a continuation marker. The preprocessor deletes the \ (and the following new-line character) and splices the physical source lines into continuous logical lines.

Except for some **#pragma** directives, preprocessor directives can appear anywhere in a program.

Macro Definition and Expansion (#define)

A *preprocessor define directive* directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

A preprocessor **#define** directive has the form:



The **#define** directive can contain an object-like definition or a function-like definition.

Object-Like Macros

An *object-like macro definition* replaces a single identifier with the specified replacement tokens. The following object-like definition causes the preprocessor to replace all subsequent instances of the identifier `COUNT` with the constant `1000` :

```
#define COUNT 1000
```

If the statement

```
int arry[COUNT];
```

appears after this definition and in the same file as the definition, the preprocessor would change the statement to

```
int arry[1000];
```

in the output of the preprocessor.

Other definitions can make reference to the identifier COUNT:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of MAX_COUNT with COUNT + 100, which the preprocessor then replaces with 1000 + 100.

If a number that is partially built by a macro expansion is produced, the preprocessor does not consider the result to be a single value. For example, the following will not result in the value 10.2 but in a syntax error.

```
#define a 10
a.2
```

Identifiers that are partially built from a macro expansion may not be produced. Therefore, the following example contains two identifiers and results in a syntax error:

```
#define d efg
abcd
```

Function-Like Macros

Function-like macro definition:

An identifier followed by a parameter list in parenthesis and the replacement tokens. The parameters are imbedded in the replacement code. White space cannot separate the identifier (which is the name of the macro) and the left parenthesis of the parameter list. A comma must separate each parameter. For portability, you should not have more than 31 parameters for a macro.

Function-like macro invocation:

An identifier followed by a list of arguments in parentheses. A comma must separate each argument. Once the preprocessor identifies a function-like macro invocation, argument substitution takes place. A parameter in the replacement code is replaced by the corresponding argument. Any macro invocations contained in the argument itself are completely replaced before the argument replaces its corresponding parameter in the replacement code.

The following line defines the macro SUM as having two parameters a and b and the replacement tokens (a + b):

```
#define SUM(a,b) (a + b)
```

This definition would cause the preprocessor to change the following statements (if the statements appear after the previous definition):

```
c = SUM(x,y);
c = d * SUM(x,y);
```

In the output of the preprocessor, these statements would appear as:

```
c = (x + y);
c = d * (x + y);
```

Use parentheses to ensure correct evaluation of replacement text. For example, the definition:

```
#define SQR(c) ((c) * (c))
```

requires parentheses around each parameter c in the definition in order to correctly evaluate an expression like:

#define

```
y = SQR(a + b);
```

The preprocessor expands this statement to:

```
y = ((a + b) * (a + b));
```

Without parentheses in the definition, the correct order of evaluation is not preserved, and the preprocessor output is:

```
y = (a + b * a + b);
```

Arguments of the # and ## operators are converted *before* replacement of parameters in a function-like macro.

The number of arguments in a macro invocation must be the same as the number of parameters in the corresponding macro definition.

Commas in the macro invocation argument list do not act as argument separators when they are:

- in character constants
- in string literals
- surrounded by parentheses

Once defined, a preprocessor identifier remains defined and in scope independent of the scoping rules of the language. The scope of a macro definition begins at the definition and does not end until a corresponding **#undef** directive is encountered. If there is no corresponding **#undef** directive, the scope of the macro definition lasts until the end of the compilation unit.

A recursive macro is not fully expanded. For example, the definition

```
#define x(a,b) x(a+1,b+1) + 4
```

expands

```
x(20,10)
```

to

```
x(20+1,10+1) + 4
```

rather than trying to expand the macro `x` over and over within itself. After the macro `x` is expanded, it is a call to function `x()`.

A definition is not required to specify replacement tokens. The following definition removes all instances of the token `debug` from subsequent lines in the current file:

```
#define debug
```

You can change the definition of a defined identifier or macro with a second preprocessor **#define** directive only if the second preprocessor **#define** directive is preceded by a preprocessor **#undef** directive. The **#undef** directive nullifies the first definition so that the same identifier can be used in a redefinition.

Within the text of the program, the preprocessor does not scan character constants or string constants for macro invocations.

Example of #define Directives

The following program contains two macro definitions and a macro invocation that refers to both of the defined macros:

CCNRAA8

```

/**
 ** This example illustrates #define directives.
 **/

#include <stdio.h>

#define SQR(s) ((s) * (s))
#define PRNT(a,b) \
    printf("value 1 = %d\n", a); \
    printf("value 2 = %d\n", b) ;

int main(void)
{
    int x = 2;
    int y = 3;

    PRNT(SQR(x),y);

    return(0);
}

```

After being interpreted by the preprocessor, this program is replaced by code equivalent to the following:

CCNRAA9

```

#include <stdio.h>

int main(void)
{
    int x = 2;
    int y = 3;

    printf("value 1 = %d\n", ( (x) * (x) ) );
    printf("value 2 = %d\n", y);

    return(0);
}

```

This program produces the following output:

```

value 1 = 4
value 2 = 3

```

RELATED REFERENCES

- “Scope of Macro Names (#undef)”
- “Operator Precedence and Associativity” on page 95
- “Parenthesized Expressions ()” on page 101

Scope of Macro Names (#undef)

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

A preprocessor **#undef** directive has the form:

```

▶▶—#—undef—identifier—————▶▶

```

If the identifier is not currently defined as a macro, **#undef** is ignored.

#undef

Example of #undef Directives

The following directives define BUFFER and SQR:

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

The following directives nullify these definitions:

```
#undef BUFFER
#undef SQR
```

Any occurrences of the identifiers BUFFER and SQR that follow these **#undef** directives are not replaced with any replacement tokens. Once the definition of a macro has been removed by an **#undef** directive, the identifier can be used in a new **#define** directive.

RELATED REFERENCES

- “Macro Definition and Expansion (#define)” on page 196

Operator

The # (single number sign) operator converts a parameter of a function-like macro into a character string literal. For example, if macro ABC is defined using the following directive:

```
#define ABC(x) #x
```

all subsequent invocations of the macro ABC would be expanded into a character string literal containing the argument passed to ABC. For example:

Invocation	Result of Macro Expansion
ABC(1)	"1"
ABC>Hello there)	"Hello there"

The # operator should not be confused with the null directive.

Use the # operator in a function-like macro definition according to the following rules:

- A parameter following # operator in a function-like macro is converted into a character string literal containing the argument passed to the macro.
- White-space characters that appear before or after the argument passed to the macro are deleted.
- Multiple white-space characters imbedded within the argument passed to the macro are replaced by a single space character.
- If the argument passed to the macro contains a string literal and if a \ (backslash) character appears within the literal, a second \ character is inserted before the original \ when the macro is expanded.
- If the argument passed to the macro contains a " (double quotation mark) character, a \ character is inserted before the " when the macro is expanded.
- The conversion of an argument into a string literal occurs before macro expansion on that argument.
- If more than one ## operator or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.
- If the result of the macro expansion is not a valid character string literal, the behavior is undefined.

Example of the # Operator

The following examples demonstrate the use of the # operator:

```
#define STR(x)      #x
#define XSTR(x)     STR(x)
#define ONE        1
```

Invocation	Result of Macro Expansion
------------	---------------------------

STR(\n "\n" '\n')	"\n \\""\n\" '\n'"
STR(ONE)	"ONE"
XSTR(ONE)	"1"
XSTR("hello")	"\"hello\""

RELATED REFERENCES

- “Null Directive (#)” on page 219
- “Function-Like Macros” on page 197
- “Macro Definition and Expansion (#define)” on page 196
- “Scope of Macro Names (#undef)” on page 199

Macro Concatenation with the ## Operator

The ## (double number sign) operator concatenates two tokens in a macro invocation (text and/or arguments) given in a macro definition.

If a macro XY was defined using the following directive:

```
#define XY(x,y)      x##y
```

the last token of the argument for x is concatenated with the first token of the argument for y.

Use the ## operator according to the following rules:

- The ## operator cannot be the very first or very last item in the replacement list of a macro definition.
- The last token of the item in front of the ## operator is concatenated with first token of the item following the ## operator.
- Concatenation takes place before any macros in arguments are expanded.
- If the result of a concatenation is a valid macro name, it is available for further replacement even if it appears in a context in which it would not normally be available.
- If more than one ## operator and/or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.

Examples of the ## Operator

The following examples demonstrate the use of the ## operator:

```
#define ArgArg(x, y)      x##y
#define ArgText(x)       x##TEXT
#define TextArg(x)       TEXT##x
#define TextText         TEXT##text
#define Jitter           1
#define bug              2
#define Jitterbug        3
```

Invocation	Result of Macro Expansion
------------	---------------------------

ArgArg(lady, bug)	"ladybug"
-------------------	-----------

Operator

Invocation	Result of Macro Expansion
ArgText(con)	"conTEXT"
TextArg(book)	"TEXTbook"
TextText	"TEXTtext"
ArgArg(Jitter, bug)	3

RELATED REFERENCES

- "Macro Definition and Expansion (#define)" on page 196

Preprocessor Error Directive (#error)

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.

The **#error** directive has the form:

```
»-#-error-_____>>
           |
           v
        character
```

Use the **#error** directive as a safety check during compilation. For example, if your program uses preprocessor conditional compilation directives, put **#error** directives in the source file to prevent code generation if a section of the program is reached that should be bypassed.

For example, the directive

```
#error Error in TESTPGM1 - This section should not be compiled
```

generates the following error message:

```
Error in TESTPGM1 - This section should not be compiled
```

RELATED REFERENCES

- "Conditional Compilation Directives" on page 213

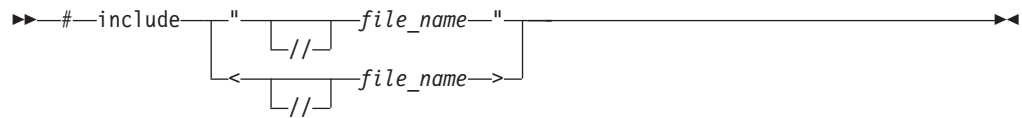
File Inclusion (#include)

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file.

A preprocessor **#include** directive has the form:

```
»-#-include-_____>>
           |
           v
          [
    <-file_name->
    <-header_name->
    identifiers
          ]
```

> z/OS In the z/OS C and C++ languages, a preprocessor **#include** directive has a different syntactical form:



z/OS You can specify a data set or an HFS file for *filename*. Use double slashes (//) before the *filename* to indicate that the file is a data set. Use a single slash (/) anywhere in the *filename* to indicate an HFS file.

In all C and C++ implementations, the preprocessor resolves macros contained in an **#include** directive. After macro replacement, the resulting token sequence must consist of a file name enclosed in either double quotation marks or the characters < and >.

For example:

```
#define MONTH <july.h>
#include MONTH
```

If the file name is enclosed in double quotation marks, for example:

```
#include "payroll.h"
```

the preprocessor treats it as a user-defined file, and searches for the file in a manner defined by the preprocessor.

If the file name is enclosed in angle brackets, for example:

```
#include <stdio.h>
```

it is treated as a system-defined file, and the preprocessor searches for the file in a manner defined by the preprocessor.

The new-line and > characters cannot appear in a file name delimited by < and >. The new-line and " (double quotation marks) character cannot appear in a file name delimited by " and ", although > can.

Declarations that are used by several files can be placed in one file and included with **#include** in each file that uses them. For example, the following file `defs.h` contains several definitions and an inclusion of an additional file of declarations:

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "mydefs.h"
```

You can embed the definitions that appear in `defs.h` with the following directive:

```
#include "defs.h"
```

In the following example, a **#define** combines several preprocessor macros to define a macro that represents the name of the C standard I/O header file. A **#include** makes the header file available to the program.

#include

```
#define C_IO_HEADER <stdio.h>

/* The following is equivalent to:
 * #include <stdio.h>
 */

#include C_IO_HEADER
```

z/OS The z/OS implementation has specially defined behavior and compiler options for include file search paths, which are described in greater detail in *z/OS C/C++ User's Guide*.

ISO Standard Predefined Macro Names

Both C and C++ provide the following predefined macro names as specified in the ISO C language standard:

Macro Name	Description
------------	-------------

<code>__DATE__</code>	<p>A character string literal containing the date when the source file was compiled.</p> <p>The value of <code>__DATE__</code> changes as the compiler processes any include files that are part of your source program. The date is in the form:</p> <p>"Mmm dd yyyy"</p>
-----------------------	--

where:

Mmm Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).

dd Represents the day. If the day is less than 10, the first d is a blank character.

yyyy Represents the year.

<code>__FILE__</code>	<p>A character string literal containing the name of the source file.</p> <p>The value of <code>__FILE__</code> changes as the compiler processes include files that are part of your source program. It can be set with the #line directive.</p>
-----------------------	--

<code>__LINE__</code>	<p>An integer representing the current source line number.</p> <p>The value of <code>__LINE__</code> changes during compilation as the compiler processes subsequent lines of your source program. It can be set with the #line directive.</p>
-----------------------	---

<code>__STDC__</code>	<p>For C, the integer 1 (one) indicates that the C compiler supports the ISO standard. If you set the language level to anything other than ANSI, this macro is undefined. (When a macro is undefined, it behaves as if it had the integer value 0 when used in a #if statement.)</p>
-----------------------	--

For C++, this macro is predefined to have the value 0 (zero). This indicates that the C++ language is not a proper superset of C, and that the compiler does not conform to ISO C. For more information on how C++ differ from ISO C, see `__STDC__` Macro in "Appendix A. C and C++ Compatibility on the z/OS Platform" on page 421.

__TIME__ A character string literal containing the time when the source file was compiled.

The value of **__TIME__** changes as the compiler processes any include files that are part of your source program. The time is in the form:

"hh:mm:ss"

where:

hh Represents the hour.

mm Represents the minutes.

ss Represents the seconds.

__cplusplus For C++ programs, this macro expands to the long integer literal 199711L, indicating that the compiler is a C++ compiler. For C programs, this macro is not defined. Note that this macro name has no trailing underscores.

RELATED REFERENCES

- "Line Control (#line)" on page 218
- "Object-Like Macros" on page 196

z/OS C/C++ Predefined Macro Names

z/OS C/C++ provides the following predefined macros. It defines the value of all these macros when you use the corresponding #pragma directive or compiler option.

Macro Name Description

__ANSI__ C Only. This macro allows only language constructs that support the ISO C standard. It is defined as 1 by using the C #pragma langlvl(ansi) directive or LONGLVL(ANSI) compiler option.

__ARCH__ This macro indicates the group number that was specified to select the instruction set for a particular machine architecture. The **__ARCH__** macro is predefined to the integer value of the ARCH suboption. For example, if you specify the ARCH(2) option, the **__ARCH__** macro is predefined to 2.

__BFP__ This macro allows Language Environment headers to map functions such as sin(x) to appropriate Language Environment calls. z/OS C/C++ sets this macro to 1 when you specify binary floating point (BFP) mode by using the FLOAT(IEEE) compiler option.

__BOOL__ C ++ Only. This macro indicates that the compiler accepts the keyword bool. The macro is predefined to the value of 1. If you specify NOKEYWORD(BOOL), the macro is not predefined.

__CHAR_SIGNED This macro indicates that the default character type is char signed. The macro is defined when the #pragma chars(signed) directive is in effect, or when the CHARS(signed) compiler option is set.

__CHAR_UNSIGNED This macro indicates that the default character type is char unsigned. The macro is defined when the #pragma chars(unsigned) directive is in effect, or when the CHARS(unsigned) compiler option is set.

__CHARSET_LIB The preprocessor defines the macro **__CHARSET_LIB** to a value of 0

#include

when you specify the NOASCII compiler option, and to a value of 1 when you specify the ASCII compiler option.

<code>__CODESET__</code>	A string literal that represents the character code set of the LOCALE compile option. If you do not use the LOCALE compile option, the macro is undefined.
<code>__COMMONC__</code>	C Only. Allows language constructs that are defined by XPG. The <code>__EXTENDED__</code> macro enables many of the constructs that <code>__COMMONC__</code> does. The compiler defines the <code>__COMMONC__</code> macro as 1 when you use the <code>#pragma langlvl(commonc)</code> directive or the <code>LANGLVL(COMMONC)</code> compile-time option.
<code>__COMPATMATH__</code>	C ++ Only. This macro indicates whether the <code>LANGLVL(OLDMATH)</code> compiler option has been specified. The macro is defined and set to 1 if the compiler option has been specified; otherwise, it is undefined. The <code>OLDMATH</code> suboption of <code>LANGLVL</code> indicates that the newer C++ function declarations are not to be introduced by the <code><math.h></code> header file.
<code>__COMPILER_VER__</code>	<p>The compiler version. The format of the version number that is provided by the macro is hex <i>PVRRMMMM</i>, where :</p> <p><i>P</i> Represents the compiler product</p> <ul style="list-style-type: none">• 0 for C/370• 1 for AD/Cycle C/370 and C/C++ for MVS/ESA• 2 for OS/390 C/C++• 4 for z/OS C/C++ Release 2 and later <p><i>V</i> Represents the version number</p> <p><i>RR</i> Represents the release number</p> <p><i>MMMM</i> Represents the modification number</p> <p>In z/OS C/C++ Version 1 Release 2, the value of the macro is <code>X'41020000'</code>.</p>
<code>__CPPUNWIND</code>	C ++ Only. The macro is set according to the compiler options <code>EXH</code> and <code>NOEXH</code> , which control whether C++ exception handling is enabled in the module being built.
<code>__DIGRAPHS__</code>	The preprocessor defines the <code>__DIGRAPHS__</code> macro to a value of 1 when you specify the <code>DIGRAPH</code> compile option. Otherwise, it is undefined.
<code>__DLL__</code>	This macro allows you to write conditional code that depends upon whether or not you have compiled your program as DLL code. For C++, the preprocessor always defines the macro as 1. For C, the preprocessor defines the macro as 1 if you specify the <code>DLL</code> compiler option. Otherwise, it is undefined.
<code>__ENUM_OPT</code>	This macro indicates that the compiler supports the <code>ENUMSIZE</code> option. It is predefined to a value of 1.
<code>__EXT</code>	This macro is used in <code>features.h</code> to control the availability of extensions to the general ISO run-time libraries. <code>__EXT</code> is defined to 1 when <code>LANGLVL(LIBEXT)</code> is specified.
<code>__EXTENDED__</code>	This macro is provided for compatibility with compilers prior to z/OS

V1R2. New program code should check the individual language feature macros (for example, the `_LONG_LONG` macro for the long long data type feature) instead of this macro.

The `__EXTENDED__` macro indicates that the compiler supports z/OS C/C++ language extensions. The compiler predefines the macro if the `#pragma langlvl (extended)` directive or the `LANGLVL(EXTENDED)` compiler option has been specified. Note that the `EXTENDED` suboption represents a group of language feature suboptions. Program code should check the individual language feature macros to determine if any of the individual language feature suboptions has been specified. Please refer to *z/OS C/C++ User's Guide* for details.

The setting of the `__EXTENDED__` macro is consistent with previous compilers only if the `EXTENDED` option is specified and no other `LANGLVL` suboptions is specified.

For example, the long long data type feature is controlled by the `LANGLVL(LONGLONG)` suboption. The `EXTENDED` suboption turns on the `LONGLONG` suboption implicitly, together with other language feature suboptions. Support for the long long data type is indicated by the `_LONG_LONG` macro.

`__FILETAG__` A string literal that represents the character code set of the `filetag` pragma associated with the current file. If no `filetag` pragma is present, the macro is undefined.

The value of `__FILETAG__` changes as the compiler processes include files that are part of your source program.

`__FUNCTION__` A character string that contains the name of the function that the z/OS C/C++ is currently compiling.

`__GOFF__` Indicates whether the `GOFF` compiler option was specified. The macro has the value 1 when the compiler option has been specified.

`__HHW_370__` Indicates that the host hardware is System/370. The preprocessor predefines this macro to a value of 1 for C and C++ compilers on System/370.

`__HOS_MVS__` Indicates that the host operating system is z/OS. z/OS C/C++ predefines this macro to have a value of 1.

`__IBMC__` C Only. This macro indicates the version number of the z/OS C compiler. The format of the version number that is provided by the macro is integer *PVRRM*, where :

- P* Represents the compiler product
 - 0 for C/370
 - 1 for AD/Cycle C/370 and C/C++ for MVS/ESA
 - 2 for OS/390 C/C++
 - 4 for z/OS C/C++ Release 2 and later

V Represents the version number

RR Represents the release number

M Represents the modification number

In z/OS C/C++ Version 1 Release 2, the value of the macro is 41020.

#include

__IBMCPP__	<p>C ++ Only. This macro indicates the version number of the z/OS C++ compiler. The format of the version number that is provided by the macro is integer <i>PVRRM</i>, where :</p> <p><i>P</i> Represents the compiler product</p> <ul style="list-style-type: none">• 0 for C/370• 1 for AD/Cycle C/370 and C/C++ for MVS/ESA• 2 for OS/390 C/C++ and 4 for z/OS C/C++• 4 for z/OS C/C++ Release 2 and later <p><i>V</i> Represents the version number</p> <p><i>RR</i> Represents the release number</p> <p><i>M</i> Represents the modification number</p> <p>In z/OS C/C++ Version 1 Release 2, the value of the macro is 41020.</p>
__IGNERRNO__	<p>The preprocessor defines the macro <code>__IGNERRNO__</code> to a value of 1 when you specify the <code>IGNERRNO</code> compile option. Otherwise, it is not defined.</p> <p>You can use this macro in header files to control the use of a <code>#pragma linkage</code> .</p>
__INITAUTO__	<p>The preprocessor defines the macro <code>__INITAUTO__</code> to the hexadecimal constant (<code>0xnnU</code>), including the parentheses, when you specify the <code>INITAUTO</code> compile option. Otherwise, it is not defined.</p> <p>You can use this macro to control source code for memory debugging. For example, you can detect values of local variables that are not explicitly set by your program.</p> <p>See the description of the <code>__INITAUTO_W__</code> macro below for information on how the compiler derives the initial byte and word value.</p>
__INITAUTO_W__	<p>The preprocessor defines the macro <code>__INITAUTO_W__</code> to the hexadecimal constant (<code>0xnnnnnnnnU</code>), including the parentheses, when you specify the <code>INITAUTO</code> compile option. Otherwise, it is not defined.</p> <p>The compiler derives the <i>effective</i> initial byte and word value as follows. If you specify a byte initializer, <i>nn</i>, then <i>nn</i> is the effective byte initial value. The effective word initial value is <i>nn</i> repeated 4 times. If you specify a word initializer, <i>nnnnnnnn</i>, the last 2 hexadecimal digits are the effective byte initial value. The effective word initial value is <i>nnnnnnnn</i>. Note that both the <code>__INITAUTO__</code> and the <code>__INITAUTO_W__</code> macros are on and off at the same time, depending on whether the <code>INITAUTO</code> option is turned on or off.</p>
__LARGE_FILES	<p>This feature test macro is used to enable large file support, which allows access to hierarchical file system (HFS) files that are larger than 2 gigabytes. The macro is defined by specifying <code>DEFINE(_LARGE_FILES)</code>, which also sets its value to 1. When an application is compiled with the <code>LANGLVL(LONGLONG)</code> compiler option and the macro is turned on, the file I/O-related functions are activated to operate on HFS files of all sizes by expanding appropriated offset and file size values to 64-bit values. (Note that</p>

the `LANGLVL(EXTENDED)` option turns on `LANGLVL(LONGLONG)` implicitly.) For further details on file I/O-related functions, see *z/OS C/C++ Run-Time Library Reference*.

`__LIBANSI__` This macro is set according to the `LIBANSI` compiler option, which means that functions should be processed with function names that match those in the ISO C library. The compiler can then assume that the behavior of these functions is that of ISO C functions.

`__LIBREL__` This macro is defined as the return value of the `librel()` library function call. The compiler calls `librel()` to determine the Language Environment library level under which it is running.

`__LOCALE__` This macro contains a string literal that represents the locale of the `LOCALE` compile option. If you do not supply a `LOCALE` compile option, the macro is undefined.

The following example illustrates how to set the run-time locale to the compile-time locale:

```
main()
{
    setlocale(LC_ALL, __LOCALE__);
    . . .
}
```

`_LONG_LONG` This macro is defined when the compiler is in a mode that permits the `long long int` and signed `long long int` data types. *z/OS C/C++* supports `long long` types by default on all language levels except ANSI. The `LONGLONG` suboption of the `LANGLVL` compiler option also sets this macro regardless of the language level. (Please refer to the `LANGLVL` option in the *z/OS C/C++ User's Guide* for support of this language feature.) When `long long` data type is available, *z/OS C/C++* defines the `_LONG_LONG` macro to 1. When the data type is unavailable, the macro is undefined. The C++ Standard Library and IBM Open Class Libraries provide additional functionality if `long long` support is turned on.

`__LONGNAME__` For C, the integer 1 indicates that you have specified the `LONGNAME` compile option or pragma. Otherwise the macro is undefined. For C++, the value of `__LONGNAME__` is always 1, even if you specify `NOLONGNAME`.

In C++, long names are always in the compilation unit. The `LONGNAME` compile option in *z/OS C++* controls whether non-C++ names will be truncated and uppercased, or left alone. You can use this option to interface with existing C code that was compiled with `NOLONGNAME`, so that the names match.

`__MI__` This macro is predefined to 1 when the `LANGLVL(LIBEXT)` suboption is specified. When the macro is defined, the machine instruction built-in functions are available. Please refer to *z/OS C/C++ Programming Guide* for a list of these functions.

This macro is also predefined by the `LANGLVL(SAA)`, `LANGLVL(SAA2)`, and `LANGLVL(COMMONC)` suboptions. Otherwise, it is not predefined.

Note that the `LANGLVL(LIBEXT)` suboption is turned on implicitly by the `LANGLVL(EXTENDED)` suboption, which means that the macro `__MI__` is also turned on indirectly by `LANGLVL(EXTENDED)`. Please refer to *z/OS C/C++ User's Guide* for details of the `LANGLVL` compiler option.

`__MVS__` This macro indicates that the host operating system is *z/OS*. It is

#include

the same as the macro `__HOS_MVS__`. For z/OS C/C++ programs, z/OS C/C++ sets this macro to 1, which indicates that you are compiling the program on z/OS.

`__OBJECT_MODEL_COMPAT__`

C ++ Only. This macro indicates that the `OBJECTMODEL (COMPAT)` option was specified. When this option is in effect, the macro is set to 1; otherwise the macro is undefined.

`__OBJECT_MODEL_IBM__`

C ++ Only. This macro indicates that the `OBJECTMODEL (IBM)` option was specified. When this option is in effect, the macro is set to 1; otherwise the macro is undefined.

`__OPTIMIZE__`

Indicates whether the `OPTIMIZE` compiler option was specified. The value of the macro is set to the level of optimization specified. The macro always has a predefined value. For example, for the `NOOPTIMIZE` or `OPTIMIZE (0)` compiler options, the macro is predefined to 0; for the `OPTIMIZE (x)`, the macro is predefined to x, where x is the value of the specified suboption.

`__RTTI_DYNAMIC_CAST__`

C ++ Only. Indicates whether the `RTTI`, `RTTI (ALL)`, or `RTTI (DYNAMICCAST)` compiler option was specified. If any of these options is specified, the macro is defined with its value set to 1. If `NORRTTI` is specified, the macro is not defined.

`__SAA__`

C Only. This macro allows only language constructs that support the most recent level of SAA C standards. It is defined as 1 by using the `#pragma langlvl (saa)` directive or `LANGLVL (SAA)` compiler option.

`__SAA_L2__`

C Only. This macro allows only language constructs that conform to SAA Level 2 C standards. It is defined as 1 by using the `#pragma langlvl (saa12)` directive or `LANGLVL (SAAL2)` compile option.

`__STRING_CODE_SET__`

This macro allows you to change the code page that the compiler uses for character string literals (character data enclosed in double quotation marks). To use this macro, you must specify it with the `DEFINE` compiler option. The following example shows you how to do this:

```
DEFINE (__STRING_CODE_SET__ = "ISO8859-1")
```

This macro affects all source files that are processed within a compilation unit, including user header files, and system header files. All string literals within a compilation unit must use the same code page. Note that you can also use the `CONVLIT` compiler option instead of this macro.

The macro does not affect the following types of string literals:

- String literals that are used in `#include` directives
- String literals that are used in `#pragma` directives
- String literals that are used to specify linkage, such as `extern "C"` (C++ only)

The following restrictions apply to this macro:

- You cannot specify this macro if you have also used predefined macros (such as `__TIMESTAMP__`) that return string literals.

`__TARGET_LIB__`

The target library version. The format of the version number provided is hex *PVRRMMMM*:

- P* Represents the z/OS C or C/C++ library product
- 0 for C/370
 - 1 for Language Environment/370 and Language Environment for MVS & VM
 - 2 for OS/390
 - 4 for z/OS Release 2 and later

V Represents the version number

RR Represents the release number

MMMM

Represents the modification number

The value of the `__TARGET_LIB__` macro depends on the setting of the `TARGET` compiler option. z/OS C/C++ sets `__TARGET_LIB__` as follows:

TARGET Suboption

`__TARGET_LIB__` Setting

zOSV1R2	0x41020000
zOsV1R1	0x41010000
OSV2R10	0x220A0000
OSV2R9	0x22090000
OSV2R8	0x22080000
OSV2R7	0x22070000
OSV2R6	0x22060000
0xnnnnnnnn	0xnnnnnnnn

`__TEMPINC__`

C++ Only. This macro indicates that the compiler is using the template-implementation file method of resolving template functions. It is defined as 1 if you are using the `TEMPINC` compile option.

`__370__`

This macro indicates that the program is compiled or targeted to run on System/370. z/OS C/C++ predefines this macro to a value of 1 for backward compatibility with earlier releases. For current programs, use `__370__`.

`__THW_370__`

This macro indicates that the target hardware is System/370. z/OS C/C++ predefines this macro to have a value of 1 for z/OS C and C++ compilers targeting System/370.

`__TIMESTAMP__`

A character string literal that contains the date and time when the source file was last modified.

The value of `__TIMESTAMP__` changes as the compiler processes any include files that are part of your source program. The date and time are in the form:

"Day Mmm dd hh:mm:ss yyyy"

where:

#include

Day	Represents the day of the week (Mon, Tue, Wed, Thu, Fri, Sat, or Sun).
Mmm	Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).
dd	Represents the day. If the day is less than 10, the first d is a blank character.
hh	Represents the hour.
mm	Represents the minutes.
ss	Represents the seconds.
yyyy	Represents the year.

This macro is available for Partitioned Data Sets (PDSs/PDSEs) and HFS source files only. For PDSE or PDS members, the ISPF timestamp for the member is used if present. For PDSE/PDS members with no ISPF timestamp, sequential datasets, or in stream source in JCL, z/OS C/C++ returns a dummy timestamp. For HFS files, z/OS C/C++ uses the system timestamp on an HFS source file. Otherwise, it returns a dummy timestamp, "Mon Jan 1 0:00:01 1990".

__TOS_MVS__	This macro indicates that the target operating system is z/OS. z/OS C/C++ predefines this macro to a value of 1.
__TUNE__	This macro specifies the processor for which the code is optimized. The value of the macro is predefined to the integer value of the TUNE suboption, which is the group number of a particular machine architecture. For example, if you specify the TUNE(2) option, the __TUNE__ macro is predefined to 2.
__XPLINK__	The preprocessor defines the macro __XPLINK__ to a value of 1 when you specify the XPLINK compiler option. Otherwise, it is not defined.

Examples of z/OS Predefined Macros

CCNX08A

```
/**
** This example illustrates the __FUNCTION__ predefined macro
** in a C program.
**/
#include <stdio.h>

int foo(int);

int main(int argc, char **argv) {
    int k = 1;
    printf (" In function %s \n",__FUNCTION__);
    foo(k);
}

int foo (int i) {
    printf (" In function %s \n",__FUNCTION__);
}
```

The output of this example is:

```
In function main
In function foo
```

CCNX08B

```

/**
 ** This example illustrates the __FUNCTION__ predefined macro
 ** in a C++ program.
 **/
#include <stdio.h>

int foo(int);

int main(int argc, char **argv) {
    int k = 1;
    printf (" In function %s \n",__FUNCTION__);
    foo(k);
}

int foo (int i) {
    printf (" In function %s \n",__FUNCTION__);
}

```

The output of this example is:

```

In function main(int, char **)
In function foo (int)

```

CCNX08C

```

/**
 ** This example illustrates the __FUNCTION__ predefined macro
 ** in a C++ program with virtual functions.
 **/
#include <stdio.h>
class X { public: virtual void func() = 0;};

class Y : public X {
    public: void func() { printf("In function %s \n", __FUNCTION__);}
};

int main() {
    Y aaa;
    aaa.func();
}

```

The output of this example is:

```

In function Y::func()

```

Conditional Compilation Directives

A *preprocessor conditional compilation directive* causes the preprocessor to conditionally suppress the compilation of portions of source code. These directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be bypassed during preprocessing. The directives are:

- **#if**
- **#ifdef**
- **#else**
- **#ifndef**
- **#elif**
- **#endif**

The preprocessor conditional compilation directive spans several lines:

- The condition specification line (beginning with **#if**, **#ifdef**, or **#ifndef**)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)

Conditional Compilation

- The **#elif** line (optional)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)
- The **#else** line (optional)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to zero (optional)
- The preprocessor **#endif** directive

For each **#if**, **#ifdef**, and **#ifndef** directive, there are zero or more **#elif** directives, zero or one **#else** directive, and one matching **#endif** directive. All the matching directives are considered to be at the same nesting level.

You can nest conditional compilation directives. In the following directives, the first **#else** is matched with the **#if** directive.

```
#ifdef MACNAME
/* tokens added if MACNAME is defined */
#   if TEST <=10
/* tokens added if MACNAME is defined and TEST <= 10 */
#   else
/* tokens added if MACNAME is defined and TEST > 10 */
#   endif
#else
/* tokens added if MACNAME is not defined */
#endif
```

Each directive controls the block immediately following it. A block consists of all the tokens starting on the line following the directive and ending at the next conditional compilation directive at the same nesting level.

Each directive is processed in the order in which it is encountered. If an expression evaluates to zero, the block following the directive is ignored.

When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

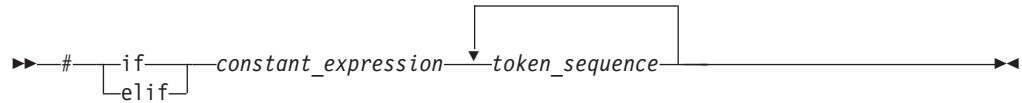
Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is a **#else** directive, the block following the **#else** directive is processed. If none of the blocks at that nesting level has been processed and there is no **#else** directive, the entire nesting level is ignored.

RELATED REFERENCES

- “**#if**, **#elif**” on page 215
- “**#ifdef**” on page 215
- “**#ifndef**” on page 216
- “**#else**” on page 216
- “**#endif**” on page 217

#if, #elif

The **#if** and **#elif** directives compare the value of *constant_expression* to zero:



If the constant expression evaluates to a nonzero value, the lines of code that immediately follow the condition are passed on to the compiler.

If the expression evaluates to zero and the conditional compilation directive contains a preprocessor **#elif** directive, the source text located between the **#elif** and the next **#elif** or preprocessor **#else** directive is selected by the preprocessor to be passed on to the compiler. The **#elif** directive cannot appear after the preprocessor **#else** directive.

All macros are expanded, any `defined()` expressions are processed and all remaining identifiers are replaced with the token 0.

The *constant_expression* that is tested must be integer constant expressions with the following properties:

- No casts are performed.
- Arithmetic is performed using **long int** values.
- The *constant_expression* can contain defined macros. No other identifiers can appear in the expression.
- The *constant_expression* can contain the unary operator **defined**. This operator can be used only with the preprocessor keyword **#if** or **#elif**. The following expressions evaluate to 1 if the *identifier* is defined in the preprocessor, otherwise to 0:

```
defined identifier
defined(identifier)
```

For example:

```
#if defined(TEST1) || defined(TEST2)
```

Note: If a macro is not defined, a value of 0 (zero) is assigned to it. In the following example, TEST must be a macro identifier:

```
#if TEST >= 1
    printf("i = %d\n", i);
    printf("array[i] = %d\n", array[i]);
#elif TEST < 0
    printf("array subscript out of bounds \n");
#endif
```

#ifdef

The **#ifdef** directive checks for the existence of macro definitions.

If the identifier specified is defined as a macro, the lines of code that immediately follow the condition are passed on to the compiler.

The preprocessor **#ifdef** directive has the form:

Conditional Compilation



The following example defines `MAX_LEN` to be 75 if `EXTENDED` is defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be 50.

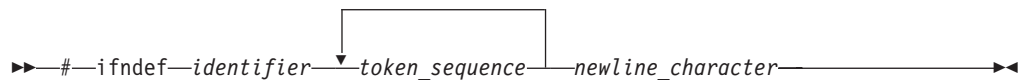
```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

`#ifndef`

The **`#ifndef`** directive checks whether a macro is not defined.

If the identifier specified is not defined as a macro, the lines of code immediately follow the condition are passed on to the compiler.

The preprocessor **`#ifndef`** directive has the form:



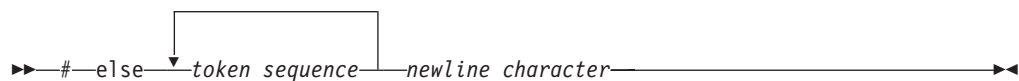
An identifier must follow the **`#ifndef`** keyword. The following example defines `MAX_LEN` to be 50 if `EXTENDED` is not defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be 75.

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

`#else`

If the condition specified in the **`#if`**, **`#ifdef`**, or **`#ifndef`** directive evaluates to 0, and the conditional compilation directive contains a preprocessor **`#else`** directive, the lines of code located between the preprocessor **`#else`** directive and the preprocessor **`#endif`** directive is selected by the preprocessor to be passed on to the compiler.

The preprocessor **`#else`** directive has the form:



RELATED REFERENCES

- “`#if`, `#elif`” on page 215
- “`#ifdef`” on page 215
- “`#ifndef`”

#endif

The preprocessor **#endif** directive ends the conditional compilation directive.

It has the form:

►► `#endif` *newline_character* ◀◀

Examples of Conditional Compilation Directives

The following example shows how you can nest preprocessor conditional compilation directives:

```

#if defined(TARGET1)
#   define SIZEOF_INT 16
#   ifdef PHASE2
#       define MAX_PHASE 2
#   else
#       define MAX_PHASE 8
#   endif
#elif defined(TARGET2)
#   define SIZEOF_INT 32
#   define MAX_PHASE 16
#else
#   define SIZEOF_INT 32
#   define MAX_PHASE 32
#endif

```

The following program contains preprocessor conditional compilation directives:

CCNRABC

```

/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/

#include <stdio.h>

int main(void)
{
    static int array[ ] = { 1, 2, 3, 4, 5 };
    int i;

    for (i = 0; i <= 4; i++)
    {
        array[i] *= 2;
    }

    #if TEST >= 1
        printf("i = %d\n", i);
        printf("array[i] = %d\n",
            array[i]);
    #endif

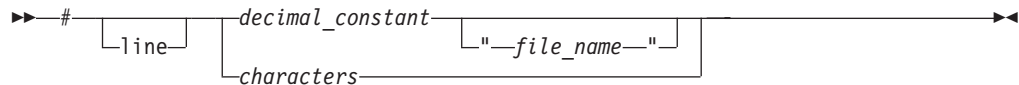
    }
    return(0);
}

```

Line Control (#line)

A *preprocessor line control directive* supplies line numbers for compiler messages. It causes the compiler to view the line number of the next source line as the specified number.

A preprocessor **#line** directive has the form:



In order for the compiler to produce meaningful references to line numbers in preprocessed source, the preprocessor inserts **#line** directives where necessary (for example, at the beginning and after the end of included text).

A file name specification enclosed in double quotation marks can follow the line number. If you specify a file name, the compiler views the next line as part of the specified file. If you do not specify a file name, the compiler views the next line as part of the current source file.

z/OS For z/OS C and C++ compilers, the *file_name* should be:

- A fully qualified sequential dataset
- A fully qualified PDS or PDSE member
- An HFS path name

z/OS The entire string is taken unchanged as the alternate source file name for the compilation unit (for example, for use by the debugger). Consider if you are using it to redirect the debugger to source lines from this alternate file. In this case, you *must* ensure the file exists as specified and the line number on the **#line** directive matches the file contents. The compiler does not check this.

In all C and C++ implementations, the token sequence on a **#line** directive is subject to macro replacement. After macro replacement, the resulting character sequence must consist of a decimal constant, optionally followed by a file name enclosed in double quotation marks.

Example of the #line Directive

You can use **#line** control directives to make the compiler provide more meaningful error messages. The following program uses **#line** control directives to give each function an easily recognizable line number:

CCNRABD

```

/**
 ** This example illustrates #line directives.
 **/

#include <stdio.h>
#define LINE200 200

int main(void)
{
    func_1();
    func_2();
}
  
```

```

}

#line 100
func_1()
{
    printf("Func_1 - the current line number is %d\n", __LINE__);
}

#line LINE200
func_2()
{
    printf("Func_2 - the current line number is %d\n", __LINE__);
}

```

This program produces the following output:

```

Func_1 - the current line number is 102
Func_2 - the current line number is 202

```

Null Directive (#)

The *null directive* performs no action. It consists of a single # on a line of its own.

The null directive should not be confused with the # operator or the character that starts a preprocessor directive.

In the following example, if MINVAL is a defined macro name, no action is performed. If MINVAL is not a defined identifier, it is defined 1.

```

#ifdef MINVAL
#
#else
#define MINVAL 1
#endif

```

RELATED REFERENCES

- “# Operator” on page 200

Pragma Directives (#pragma)

A *pragma* is an implementation-defined instruction to the compiler. It has the general form:



where *character_sequence* is a series of characters giving a specific compiler instruction and arguments, if any.

The *character_sequence* on a pragma is not subject to macro substitutions.

More than one pragma construct can be specified on a single **#pragma** directive. The compiler ignores unrecognized pragmas.

z/OS Pragma Directives

z/OS This section describes the pragma directives that are specific to the z/OS platform.

The *character_sequence* on a pragma is not subject to macro substitutions, unless otherwise stated.

You can specify more than one pragma construct on a single #pragma directive. The compiler ignores unrecognized pragmas.

The z/OS C/C++ compiler recognizes the following pragmas:

chars	Sets the sign type of character data.
checkout	Controls the diagnostic messages that are generated by the z/OS C compiler CHECKOUT option, and the z/OS C++ compiler INFO option.
comment	Places a comment into the object module. Under some circumstances it places the comment in the load module as well. This pragma must appear before any z/OS C or C++ code.
convlit	Provides a means for changing the assumed code page for character string literals.
csect	Identifies the name for either the code, static, or test control section (CSECT). The IPA Link step does not use this name; it uses CSECT names that are specified in the IPA control file.
define	C++ Only. This pragma forces the definition of a template class without actually defining an object of the class.
disjoint	This pragma lists the identifiers that are not aliased to each other within the scope of their use.
enum	This z/OS pragma specifies the amount of storage occupied by enumerations.
environment	C Only. Use z/OS C code as an assembler substitute.
export	Declares that an external function or variable is to be exported.
filetag	Specifies the code set in which the source code was entered.
hdrstop	This pragma is accepted and ignored. It must appear before any code.
implementation	C++ Only. This pragma tells the compiler the name of the file that contains the function template definitions. These definitions correspond to the template declarations in the include file that contains the pragma.

#pragma

info	C++ Only. This pragma controls the diagnostic messages that are generated by the INFO compiler option.
inline	C Only. This pragma specifies that a C function is to be inlined.
isolated_call	Lists functions that do not alter data objects visible at the time of the function call.
langlvl	Selects the z/OS C language level for compilation.
leaves	Specifies that a named function never returns to the instruction following the call to that function. The library function longjmp is an example of such a function. Note that this pragma directive provides more freedom to the optimizer. Under the right conditions, it might aggressively optimize the code following the call site.
linkage	C Only. This pragma identifies the linkage or calling convention that is used on a function call.
longname	Specifies that the compiler is to generate not-truncated and mixed case names in the object module that is produced by the compiler. It must appear before any code.
map	Tells the compiler to convert all references to an identifier to a new name.
margins	Specifies the columns in the input line to scan for input to the compiler.
namemangling	C++ Only. Specifies the maximum length for the external symbol names that are generated from C++ source code.
noinline	Specifies that a z/OS C or C++ function is not to be inlined.
object_model	C++ Only. Specifies the object model to use for the structures, unions, and classes that follow it.
options	C Only. This pragma specifies options to the compiler in your source program.
option_override	Directs the compiler to optimize functions at different optimization levels than the one specified on the command line by the OPTIMIZE option. With this pragma directive, you can leave specified functions unoptimized, while optimizing the rest of your application. This eases the debugging effort of functions that are problematic under optimization, by allowing you to isolate those functions.
pack	Specifies the alignment rules to use for the structures, unions, and classes that follow it.
page	C Only. This pragma skips pages of the generated source listing.
pagesize	C Only. This pragma sets the number of lines per page for the generated source listing.

#pragma

priority	C++ Only. This pragma specifies the order in which z/OS C/C++ initializes static objects at run time.
reachable	Specifies that you can reach the instruction after a specified function from a point in the program other than the return statement in the named function. The library function, setjmp, is an example of such a function.
report	C++ Only. Allows you to specify the minimum severity level (type) of report messages that are displayed or whether specific messages are enabled or disabled.
runopts	Specifies a list of run-time options for z/OS C/C++ to use at execution time.
sequence	Defines the section of the input line that is to contain sequence numbers.
skip	C Only. This pragma skips lines of the generated source listing.
strings	Sets storage type for strings.
subtitle	C Only. This pragma places text on generated source listings.
target	C Only. This pragma specifies the operating system or run-time environment for which z/OS C/C++ creates the object module. It must appear before any z/OS C code.
title	C Only. This pragma places text on generated source listings.
variable	Specifies that z/OS C/C++ is to use the named object in a reentrant or non-reentrant fashion.
wsiz eof	Specifies the behavior of the sizeof operator either to that prior to the OS/390 C/C++ Version 1 Release 3 compilers, or to the z/OS C/C++ compiler.

Restrictions on z/OS #pragma Directives

z/OS If you have any pragmas that are not common to both C and C++ in code that will be compiled by both compilers, you may add conditional compilation directives around the pragmas. This is not strictly necessary since unrecognized pragmas are ignored.

For example, #pragma object_model is only recognized by the C++ compiler, so you may decide to add conditional compilation directives around the pragma.

```
#ifdef __cplusplus
#pragma object_model(pop)
#endif
```

The following table lists the restrictions on using #pragma directives, and shows whether a directive is valid in z/OS C, C++, or both. A blank entry in the table indicates no restrictions.

Table 7. Restrictions on #pragmas

#pragma	Restriction on Number of Occurrences	Restriction on Placement	C	C++
chars	Once.	On the first #pragma directive, and before any code or directive, except for the pragmas filetag, longname, langlvl or target, which may precede this directive.	yes	yes
checkout			yes	yes
comment	The copyright directive can appear only once.	The copyright directive must appear before any z/OS C or C++ code.	yes	yes
convlit			yes	yes
csect	Three times. Once for code, once for static data, and once for debug information.		yes	yes
define		Wherever a declaration is allowed.		yes
disjoint		Wherever a declaration is allowed.	yes	yes
enum			yes	yes
environment			yes	
export		Cannot export the main() function.	yes	yes
filetag	Once per file scope.	On the first #pragma directive, and before any code or directive, except for all conditional compilation directives (such as #if or #ifdef) which may precede this directive.	yes	yes
implementation		Wherever a declaration is allowed.		yes
info				yes
inline		At file scope.	yes	
isolated_call		Wherever a declaration is allowed.	yes	yes
langlvl	Once	Before any C code	yes	
leaves			yes	yes
linkage	Can appear more than once for each function, as long as one #pragma does not contradict another #pragma.		yes	
longname	Once.	On the first #pragma directive, except for pragmas filetag, chars, langlvl or target, which may precede this directive.	yes	yes
map			yes	yes
margins			yes	yes
namemangling				yes
noinline		At file scope.	yes	yes
object_model				yes
options		Before any z/OS C code.	yes	
option_override			yes	yes
pack			yes	yes
page			yes	
pagesize			yes	

#pragma

Table 7. Restrictions on #pragmas (continued)

#pragma	Restriction on Number of Occurrences	Restriction on Placement	C	C++
priority				yes
reachable			yes	yes
report				yes
runopts			yes	yes
sequence			yes	yes
skip			yes	
strings	Once.	Before any z/OS C or C++ code.	yes	yes
subtitle			yes	
target	Once.	On the first #pragma directive, and before any code or directive, except for pragmas filetag, chars, langlvl, or longname, which may precede this directive.	yes	
title			yes	
variable			yes	yes
wsizEOF			yes	yes

IPA Considerations

z/OS Interprocedural Analysis (IPA), through the IPA compiler option, is a mechanism for performing optimizations across the compilation units of your z/OS C or C++ program. IPA also performs optimizations not otherwise available with the z/OS C/C++ compiler.

Many #pragma directives do not have any special behavior under IPA. They have the same effect on a program compiled with or without the IPA option.

You may see changes during the IPA Link step, due to the effect of a #pragma directive. The IPA Link step detects and resolves the conflicting effects of #pragma directives, and the conflicting effects of #pragma directives and compiler options that you specified for different compilation units. There may also be conflicting effects between #pragma directives and equivalent compiler options that you specified for the IPA Link step.

IPA resolves these conflicts similar to the way it resolves conflicting effects of compiler options that are specified for the IPA Compile step and the IPA Link step. The Compiler Options Map section of the IPA Link step listing shows the conflicting effects between compiler options and #pragma directives, along with the resolutions.

For those #pragma directives where there are special considerations for IPA, the following #pragma descriptions include IPA-related information.

chars

z/OS The z/OS #pragma chars directive specifies that the compiler is to treat all char objects as signed or unsigned.

```

>> #pragma chars ( ( unsigned ) ) ----->>
                   |
                   | signed

```

This pragma must appear on the first #pragma directive. It must also appear before any code or directive, except for the pragmas `filetag`, `longname`, `langlvl` or `target`. These pragmas may precede this directive. Once specified, it applies to the rest of the file and you cannot turn it off. If a source file contains any functions that you want to compile without #pragma chars, place these functions in a different file.

The default character type behaves like an unsigned char.

checkout

z/OS The z/OS #pragma checkout directive is a z/OS C/C++ directive and an addition to the SAA Standard.

This pragma can appear anywhere that a preprocessor directive is valid.

```

>> #pragma checkout ( ( resume ) ) ----->>
                       |
                       | suspend

```

With #pragma checkout, you can suspend the diagnostics that the CHECKOUT C compiler option or the INF0 C++ compiler option performs during specific portions of your program. You can then resume the same level of diagnostics later in the file.

Nested #pragma checkout directives are allowed and behave as the following example demonstrates:

```

/* Assume CHECKOUT (PPTRACE) had been specified */
#pragma checkout(suspend) /* No CHECKOUT diagnostics are performed */
...
#pragma checkout(suspend) /* No effect */
...
#pragma checkout(resume) /* No effect */
...
#pragma checkout(resume) /* CHECKOUT (PPTRACE) diagnostics continue */

```

comment

z/OS The z/OS #pragma comment directive places a comment into the object module. This pragma must appear before any C or C++ code or directive in a source file. The "token_sequence" field in this pragma has a 1024-byte limit.

```

>> #pragma comment ( ( compiler
                     |
                     | date
                     |
                     | timestamp
                     |
                     | copyright
                     |
                     | user ,—"token_sequence—" ) ) ----->>

```

The comment type can be:

compiler The compiler appends its name and version in an END information record at the end of the generated object module. z/OS C/C++

#pragma

does not include the name and version when it generates an executable, nor does it load the name and version into memory when it runs the program. This information can be printed out using the C370LIB utility with the MAP option.

date The compiler appends the date and time of compilation in an END information record at the end of the generated object module. z/OS C/C++ does not include the date and time when it generates an executable nor does it load the date and time into memory when it runs the program. This information can be printed out using the C370LIB utility with the MAP option.

timestamp The compiler appends the date and time of the last modification of the source in an END information record at the end of the generated object module. z/OS C/C++ does not include the date and time when it generates an executable nor does it load the date and time into memory when it runs the program. This information can be printed out using the C370LIB utility with the MAP option.

If z/OS C/C++ cannot find the timestamp for a source file, the #pragma comment directive returns Mon Jan 1 0:00:01 1990.

copyright The compiler places text that is specified by the *token_sequence*, if any, into the generated object module. When z/OS C/C++ creates an executable, it includes the *token_sequence* in the load module. The module is loaded into memory when z/OS C/C++ runs the program. In addition to the restriction that the directive be the first instruction in a source file, the copyright directive can appear only once in a compilation unit, whereas the other types of comment directives can appear more than once.

user The compiler places the text that is specified by the *token_sequence*, if any, into the generated object module. When z/OS C/C++ creates an executable, the *token_sequence* is included in the load module. Note that z/OS C/C++ does *not* necessarily load it into memory when it runs the program. z/OS C/C++ places the *token_sequence* on END records in columns 34 to 71.

The characters in the *token_sequence* field, if specified, must be enclosed in double quotation marks (").

You can display the object-file comments by using the MAP option for the C370LIB utility.

IPA Considerations for the #pragma comment

The #pragma comment directive affects the IPA Compile step only if the OBJECT suboption of the IPA compile option is in effect. With the IPA(OBJONLY) option, the pragma has the same effect as if IPA were not specified.

During the partitioning process in the IPA Link step, the compiler places the text string information #pragma comment at the beginning of partition 0. Partition 0 is the initialization partition.

convlit

z/OS The z/OS #pragma convlit directive allows you to suspend the string literal conversion that the convlit compiler option performs during specific portions of your program. You can then resume the conversion at some later point in the file.

```

▶▶ #pragma convlit ( resume suspend ) ▶▶

```

The pragma is effective only when you specify the CONVLIT compile option.

If you select the PPONLY option, z/OS C/C++ echoes the convlit pragma to the expanded source file.

You can nest #pragma convlit directives. They behave as the following example demonstrates:

```

/* Assume CONVLIT (<codepage>) had been specified */
#pragma convlit(suspend) /* No string literal conversion */
...
#pragma convlit(suspend) /* No effect */
...
#pragma convlit(resume) /* No effect */
...
#pragma convlit(resume) /* String literal conversion continues */

```

Macros, user-defined and pre-defined, are replaced before tokenization; therefore, using #pragma convlit(suspend) and #pragma convlit(resume) around a macro definition would have no effect.

For example:

```

/* No effect on macro definition when using #pragma convlit(suspend)
   and #pragma convlit(resume)*/

main() {
    #pragma convlit (suspend)

    #define str "Hello World!"
    puts(str);          /* macro str is not converted */

    #pragma convlit(resume)

    puts(str);          /* macro str is converted */
}

```

csect

▶ z/OS The z/OS #pragma csect directive identifies the name for either the code, static, or debug control section (CSECT).

```

▶▶ #pragma csect ( CODE STATIC TEST , " name " ) ▶▶

```

It is a z/OS C/C++ specific pragma, and an addition to the SAA Standard.

code Specifies the CSECT that contains the executable code (C functions) and constant data.

static Designates the CSECT that contains all program variables with the static storage class and all character strings.

test Designates the CSECT that contains debug information. You must specify the TEST option.

#pragma

The above syntax encloses the *name* in double quotation marks. This is the name that is used for the applicable CSECT (code, static, or test). z/OS C/C++ does not map the name in any way, including uppercasing. If the name is greater than 8 characters, you must turn on the LONGNAME option and use the binder. The name must not conflict with the name of an exposed name (external function or object) in a source file. In addition, it must not conflict with another #pragma csect directive or #pragma map directive. For example, the name of the code CSECT must differ from the name of the static and test CSECTs.

At most, three #pragma csect directives can appear in a source program as follows:

- One for the code CSECT
- One for the static CSECT
- One for the debug CSECT

Consider the case in which there is no #pragma csect directive in the source file and you specify the CSECT compiler option. In this case, z/OS C/C++ automatically generates CSECT names from the source file name. For examples that show the file names that are generated when using either the #pragma csect or the CSECT compiler option, see the section that describes the CSECT option in the *z/OS C/C++ User's Guide*.

When both #pragma csect and the CSECT compiler option are specified, the compiler first uses the option to generate the csect names, and then the #pragma csect overrides the names generated by the option. Suppose that you compile the following code with the option CSECT(abc) and program name foo.c.

```
#pragma csect (STATIC, "blah")
int main ()
{
    return 0;
}
```

First, the compiler generates the following csect names:

```
STATIC: abc#foo#S
CODE: abc#foo#C
TEST: abc#foo#T
```

Then the #pragma csect overrides the static CSECT name, which renders the final CSECT name to be:

```
STATIC: blah
CODE: abc#foo#C
TEST: abc#foo#T
```

Private code has a disadvantage. When new code is linked to an executable that contains old code, the new code replaces the old. The old code, however, is not discarded from the executable. The size of the executable will grow, and you may get duplicates of functions. Naming the CSECTs with this directive replaces the old code with the new, and removes the old code from the executable. If you want replacement and removal, name the code, static, and test CSECT.

IPA Considerations for the #pragma csect

Use the #pragma csect directive when naming regular objects only if the OBJECT suboption of the IPA compiler option is in effect. Otherwise, the compiler discards the CSECT names that #pragma csect generated. With the IPA(OBJONLY) option, the pragma has the same effect as if the IPA option were not specified.

define (z/OS C++ Only)

z/OS The z/OS #pragma define directive forces the definition of a template class without actually defining an object of the class.

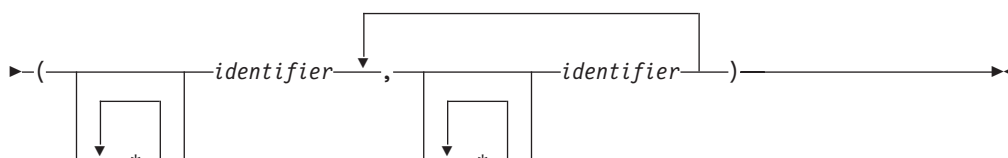
►► #pragma define (—*template_class_name*—) ◀◀

The pragma can appear anywhere that a declaration is allowed. Use the pragma to organize your program to efficiently or automatically generate template functions.

disjoint

z/OS The z/OS #pragma disjoint directive lists the identifiers that are not aliased to each other within the scope of their use. In the following syntax diagram, *identifier* is the name of a variable:

►► #pragma disjoint ◀◀



The directive informs the compiler that none of the identifiers listed shares the same physical storage, which provides more opportunity for optimizations. If any identifiers actually share physical storage, the pragma may give incorrect results.

The pragma can appear anywhere in the source program that a declaration is allowed. An identifier in the directive must be visible at the point in the program where the pragma appears. The identifiers in the disjoint name list cannot refer to any of the following:

- A member of a class, structure, or union
- A structure, union, or enumeration tag
- An enumeration constant
- A typedef name
- A label

You must declare the identifiers before using them in the pragma. Your program must not dereference a pointer in the identifier list nor use it as a function argument before it appears in the directive.

The following example shows the use of #pragma disjoint.

```

int a, b, *ptr_a, *ptr_b;

#pragma disjoint(*ptr_a, b) /* *ptr_a never points to b */
#pragma disjoint(*ptr_b, a) /* *ptr_b never points to a */
one_function()
{
    b = 6;
}

```

#pragma

```

    *ptr_a = 7;    /* Assignment will not change the value of b */

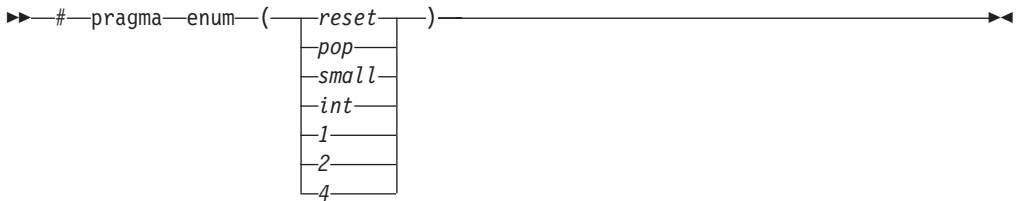
    another_function(b);    /* Argument "b" has the value 6 */
}

```

External pointer `ptr_a` does not share storage with and never points to the external variable `b`. Consequently, assigning 7 to the object to which `ptr_a` points will not change the value of `b`. Likewise, external pointer `ptr_b` does not share storage with and never points to the external variable `a`. The compiler can assume that the argument to `another_function` has the value 6 and will not reload the variable from memory.

enum

z/OS The z/OS #pragma enum directive specifies the amount of storage occupied by enumerations. The directive is in effect until the next valid #pragma enum directive is encountered. The syntax of the option is as follows:



A `#pragma enum(reset)` or `#pragma enum(pop)` resets the enum setting to the one that was in effect before the current setting. If no previous enum setting was specified in the file, the one specified with the `ENUM` compiler option is used. For every `#pragma enum` directive in your program, it is good practice to have a corresponding `#pragma enum(reset)` or `#pragma enum(pop)` as well. This is the only way to prevent one file from potentially changing the enum setting of another file that is included.

The default is `#pragma enum(small)`. It allocates to an enum variable the amount of storage that is required by the smallest predefined type that can represent that range of enum constants. The other suboptions allocate a specific amount. If the specified storage size is smaller than that required by the range of enum constants, the C compiler issues an error. For C++, a suboption that can represent the range will be used, and a warning is issued.

For example:

```
#pragma enum(1)
enum e_tag {
    a=0,
    b=SHRT_MAX /* error CBC3387 */
} e_var;
#pragma enum(reset)
```

For C++, a warning message is issued, and a suboption that can represent the range of enum constants will be used.

The following table summarizes the `#pragma enum` suboptions:

reset	Sets the enum setting to that which was in effect before the current setting.
-------	---

#pragma

pop	Restores the enum setting to that which was in effect immediately before the current setting.
small	Specifies that an enumeration occupies a minimum amount of storage: either 1, 2, or 4 bytes of storage, depending on the range of the enum constants.
int	Specifies that an enumeration occupies 4 bytes of storage and are represented by int.
1	Specifies that an enumeration occupies 1 byte of storage.
2	Specifies that an enumeration occupies 2 bytes of storage.
4	Specifies that an enumeration occupies 4 bytes of storage.

You cannot have #pragma enum directives within the declaration of an enum. The following code segment generates a warning message and the second occurrence of the enum option is ignored:

```
#pragma enum(small)
enum e_tag {
    a,
    b,
#pragma enum(int) /*cannot be within a declaration */
    c
} e_var;
```

The following table illustrates the preferred sign and type for each range of enum constants:

Table 8. ENUM Constants for C and C++

ENUM Constants	small	1	2	4	int
0..127	unsigned char	signed char	short	int	int
-128..127	signed char	signed char	short	int	int
0..255	unsigned char	unsigned char	short	int	int
0..32767	unsigned short	ERROR	short	int	int
-32768..32767	short	ERROR	short	int	int
0..65535	unsigned short	ERROR	unsigned short	int	int
0..2147483647	unsigned int	ERROR	ERROR	int	int
-(2147483647+1)..2147483647	int	ERROR	ERROR	int	int
0..4294967295	unsigned int	ERROR	ERROR	unsigned int	unsigned int

environment (z/OS C Only)

z/OS The z/OS #pragma environment directive is specific to z/OS C and an addition to the SAA Standard.

►► #pragma environment (—function—, nolib) —————►►

With the #pragma environment directive, you can use z/OS C code as an assembler substitute. The directive allows you to do the following:

- Specify entry points other than main
- Omit setting up a C environment on entry to this function

#pragma

- Specify several system exits that are written in z/OS C code in the same executable

If you specify `no1ib`, the environment is established, and the library is not loaded at run time. If you do not specify anything, the library is loaded.

Note: If you specify any other value than `no1ib` after the function name, behavior is not defined.

export

z/OS The z/OS `#pragma export` directive declares that a function or variable is to be exported. It also specifies the name of the function or variable to be referenced outside the module. You can use this `#pragma` to export functions or variables from a DLL module.

► `#pragma export (function | variable)` ◀

`#pragma export` is specific to z/OS C/C++ and an addition to the SAA standard.

With the `#pragma export` directive, you can export specific functions and variables to the users of your DLL.

You can specify this pragma anywhere in the DLL source code, on its own line, or with other pragmas. You can also specify it before or after the definition of the variable or function. You must externally define the exported function or variable.

If the specification for a `const` variable in a `#pragma export` directive conflicts with the `ROCONST` option, the pragma directive takes precedence over the compile option, and the compiler issues an informational message. The `const` variable gets exported and it is considered re-entrant.

Note: You cannot export the `main()` function. You can also use the `_Export` keyword to export a function.

IPA Considerations for the #pragma export

If you specify this `#pragma` in your source code in the IPA Compile step, you cannot override the effects of this `#pragma` on the IPA Link step.

filetag

z/OS The z/OS `#pragma filetag` directive specifies the code set in which the source code was entered.

► `#pragma filetag (—"code set name"—)` ◀

Since the `#` character is variant between code sets, use the trigraph representation `??=` instead of `#` as illustrated below.

The `#pragma filetag` directive must appear at most once per source file. It must appear before the first statement or directive, except for all conditional compilation directives, which may precede this directive. For example:

```

??=ifdef COMPILER_VER          /* This is allowed. */
    ??=pragma filetag ("code set")

??=endif

```

It should not appear in combination with any other #pragma directives. For example, the directive is incorrect:

```

??=pragma filetag ("IBM-1047") export (baffle_1)

```

If there are comments before the pragma, z/OS C/C++ does not translate them to the code page that is associated with the LOCALE option.

implementation (z/OS C++ Only)

z/OS The z/OS #pragma implementation directive tells the compiler the name of the file containing the function-template definitions. These definitions correspond to the template declarations in the include file which contains the pragma.

```

>> #pragma implementation (—string_literal—) <<

```

This pragma can appear anywhere that a declaration is allowed. Use this pragma to organize your program to efficiently or automatically generate template functions.

#pragma implementation is only effective if the TEMPINC option is in effect. When the TEMPLATEREGISTRY option is specified, #pragma implementation has no meaning and is ignored. Note that the TEMPINC and TEMPLATEREGISTRY options are mutually exclusive.

If the NOTEMPINC option is in effect, you must test the value of the __TEMPINC__ macro, and conditionally include the required source.

info (z/OS C++ Only)

z/OS The z/OS #pragma info directive controls the diagnostic messages that are generated by the INFO compiler option.

```

>> #pragma info (—suspend—) <<
                    resume

```

You can use this pragma directive in place of the INFO option.

Use #pragma info suspend to suspend the diagnostics that the INFO compiler option performs during specific portions of your program. You can then use #pragma info resume to resume the same level of diagnostics later in the file.

You can also use #pragma checkout to suspend or resume diagnostics.

#pragma

inline (z/OS C Only)

z/OS The z/OS #pragma inline directive specifies whether or not the *function* is to be inlined. The pragma can be anywhere in the source, but must be at file scope. #pragma inline has no effect if you have not specified the INLINE or the OPT compiler option.

→ #pragma { inline | noinline } (—function—) →

The #pragma inline directive is specific to z/OS C and is an addition to the SAA Standard.

The #pragma noinline directive is specific to z/OS C and C++, and is an addition to the SAA Standard.

If you specify #pragma inline, the function is inlined on every call. The function is inlined in both selective (NOAUTO) and automatic (AUTO) mode. For z/OS C++, you can inline functions using the inline keyword.

If you specify #pragma noinline in your C or C++ program, the function is never inlined when you call it. This pragma has no effect when you specify NOAUTO with the z/OS C/C++ INLINE compiler option.

The default when compiling with the OPTIMIZE option is to inline functions even if the z/OS C++ inline keyword has not been specified. The default when compiling with the NOOPTIMIZE option is to only inline C++ functions that are:

- Implicitly inlined; that is when the code for a member function is included inside a class definition
- Explicitly inlined; that is when the inline keyword is used when declaring a function

For C, you can place the #pragma inline and noinline directives anywhere in the source. They must be at file scope.

The #pragma noinline directive is the only way to turn off inlining of functions that have been implicitly or explicitly inlined. It also takes precedence over the z/OS C++ inline keyword.

IPA Considerations for the #pragma inline

The compiler uses the IPA Link control file directive in the following cases:

- If you specify both the #pragma noinline directive and the IPA Link control file inline directive for a function
- If you specify both the #pragma inline directive and the IPA Link control file noinline directive for a function

Example CCNRABE

```
/* this example shows how #pragma inline may be used */
```

```
#pragma csect(code,"MYCFIL")
#pragma csect(static,"MYSFIL")
#pragma options(INLINE)
```

```
#include <stdio.h>
```

```

#include <stdlib.h>

static int (writerecord) (int, char *);

#pragma inline (writerecord)

int main()
{
    int chardigit;
    int digit;

    printf("Enter a digit\n");
    chardigit = getchar();

    digit = chardigit - '0';
    if (digit < 0 || digit > 9)
    {
        printf("The digit you entered is not between 1 and 8\n");
        exit(99);
    }

    switch(digit)
    {
        case 0:
            writerecord(0, "entered 0");
            break;
        case 1:
            writerecord(1, "entered 1");
            break;
        default:
            writerecord(9, "entered other");
    }
}

static int writerecord (int digit, char *phrase)
{
    switch (digit)
    {
        case 0:
            printf("writerecord 0: ");
            printf("%s\n", phrase);
            break;
        case 1:
            printf("writerecord 1: ");
            printf("%s\n", phrase);
            break;
        case 2:
            printf("writerecord 2: ");
            printf("%s\n", phrase);
            break;
        case 3:
            printf("writerecord 3: ");
            printf("%s\n", phrase);
            break;
        default:
            printf("writerecord X: ");
            printf("%s\n", phrase);
    }
}

return 0;
}

```

#pragma isolated_call

z/OS The z/OS #pragma isolated_call directive lists functions that do not alter data objects visible at the time of the function call. In the following syntax diagram, *identifier* is a primary expression that can be an identifier, operator function, conversion function, or qualified name:



The pragma must appear before calls to the functions in the identifier list. You must declare the identifiers that are listed before using them in the pragma. They must be of type function, or a typedef of function. If a name refers to an overloaded function, all variants of that function declared before the pragma are marked as isolated calls.

The pragma informs the compiler that none of the functions listed has side effects. For example:

- Accessing a volatile object
- Modifying an external object
- Modifying a file

Otherwise, you can consider calling a function that does any of the above to be side effects.

Consider any change in the state of the run-time environment a side effect. Passing function arguments by reference is one side effect that z/OS C/C++ allows. In general, however, functions with side effects can give incorrect results when listed in #pragma isolated_call directives.

Marking a function as isolated indicates to the optimizer that external and static variables cannot be changed by the called function. It also indicates that references to storage can be deleted from the calling function where appropriate. Do not specify a function that calls itself or relies on local static storage. Listing such functions in the #pragma isolated_call directive can give unpredictable results.

When a function is marked as isolated, the compiler can make more optimistic assumptions about what variables the function modifies. The compiler may move function calls to functions that are flagged as isolated to a different location in the code or even remove them entirely.

The following example routines shows you when to use the #pragma isolated_call directive (routine addmult). It also shows you when not to use it (routines same and check):

```
#include <stdio.h>
#include <math.h>

int addmult(int op1, int op2);
#pragma isolated_call(addmult)

/* This routine is a good candidate to be flagged as isolated as its */
/* result is constant with constant input and it has no side effects. */
int addmult(int op1, int op2) {
    int rslt;
```

```

    rslt = op1*op2 + op2;
    return rslt;
}

/* The routine 'same' should not be flagged as isolated as its state */
/* (the static variable delta) can change when it is called. */
int same(double op1, double op2) {
    static double delta = 1.0;
    double temp;

    temp = (op1-op2)/op1;
    if (fabs(temp) < delta)
        return 1;
    else {
        delta = delta / 2;
        return 0;
    }
}

/* The routine 'check' should not be flagged as isolated as it has a */
/* side effect of possibly emitting output. */
int check(int op1, int op2) {
    if (op1 < op2)
        return -1;
    if (op1 > op2)
        return 1;
    printf("Operands are the same.\n");
    return 0;
}

```

IPA Considerations for the #pragma isolated_call

If you specify this #pragma in your source code in the IPA Compile step, you cannot override the effects of this #pragma on the IPA Link step.

langlvl

C The z/OS #pragma langlvl directive selects the C language level for compilation.

```

>> #pragma langlvl (ansi | common | extended | saa | saa12) <<

```

You can only specify this pragma only once in a source file. It must appear before any statements in a source file. The compiler uses predefined macros in the header files to make declarations and definitions available that define the specified language level. When both the pragma and the compiler option are specified, the compiler option takes precedence over the pragma.

The default language level is EXTENDED.

ansi	Defines the predefined macros <code>__ANSI__</code> and <code>__STDC__</code> and undefines other langlvl variables. It allows only language constructs that support the ISO C standards.
extended	Defines the predefined macro <code>__EXTENDED__</code> and undefines other langlvl variables. The default language level is EXTENDED. z/OS C defines the <code>__EXTENDED__</code> macro as 1. Note that #pragma

#pragma

`langlvl(EXTENDED)` has no effect in the z/OS UNIX environment. In z/OS UNIX, you must use the compile option `LANGVL(EXTENDED)` instead of the pragma.

<code>commonc</code>	Defines the predefined macro <code>__COMMONC__</code> and <code>__EXTENDED__</code> and undefines other <code>langlvl</code> variables. This language level allows compilation of code that contains constructs defined by the X/Open Portability Guide (XPG) Issue 3 C language (referred to as Common Usage C). It is roughly equivalent to what is commonly known as K&R C.
<code>saa</code>	Defines the predefined macro <code>__SAA__</code> and undefines other <code>langlvl</code> variables.
<code>saa12</code>	Defines the predefined macro <code>__SAA_L2__</code> and undefines other <code>langlvl</code> variables.

The `#pragma langlvl(extended)` permits packed decimal types and it issues a warning message when it detects assignment between integral types and pointer types.

The `#pragma langlvl(ansi)` does not permit packed decimal types and issues an error message when it detects assignment between integral types and pointer types.

The `LANGVL` compiler option has the same effect as this pragma.

leaves

z/OS The z/OS `#pragma leaves` directive takes a function name, and specifies that the function never returns to the instruction following a call to that function.

```
→ #pragma leaves ( function_name ) →
```

When enabled, the `leaves` pragma provides information to the compiler that enables it to explore additional opportunities for optimization. Also see “reachable” on page 252.

When you specify the `LIBANSI` compiler option, you tell the compiler that ANSI C library function names refer to ANSI C library functions. These function names do not refer to your own version of the library functions, which may have different semantics. In this case, the compiler checks whether the `longjmp` family of functions (`longjmp`, `_longjmp`, `siglongjmp`, and `_siglongjmp`) contain `#pragma leaves`. If the functions do not contain this pragma directive, the compiler will insert this directive for that function. This is not shown in the listing.

IPA Considerations for the #pragma leaves

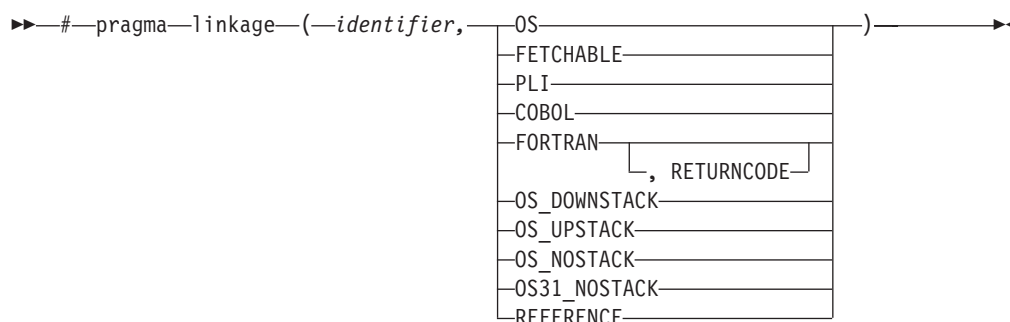
If you specify the `#pragma leaves` directive in your source code in the IPA compile step, you cannot override the effects of this directive in the IPA link step.

Effect on IPA Link Step

The leave (and reachable) status set by the IPA compile step remains in effect and cannot be unset by option settings during the IPA Link Step. (For more information, refer to “reachable” on page 252 as well.)

linkage (z/OS C only)

z/OS The z/OS #pragma linkage directive identifies the entry point of modules that are used in interlanguage calls.



The *identifier* either identifies the name of the function that is to be the entry point of the module. Or, it identifies a typedef that will be used to define the entry point.

In z/OS C++, you accomplish this by using `extern "linkage-type"` when declaring an identifier, for example,

```
extern "FORTRAN" void f();
extern "COBOL" void g();
```

The #pragma linkage directive also designates other entry points within a program that you can use in a fetch operation.

The following are the linkage entry points:

FETCHABLE	Specifies a name, other than main, as an entry point within the program. This pragma also indicates that this name (<i>identifier</i> in the syntax diagram) can be used in a <code>fetch()</code> operation.
OS	Designates an entry point (<i>identifier</i> in the syntax diagram) as an OS linkage entry point. OS linkage is the basic linkage convention that is used by the operating system. If the caller is compiled with NOXPLINK, on entry to the called routine it registers 13 points to a standard Language Environment stack frame, beginning with a 72-byte save area (which is compatible with Language Environment languages that expect by-reference calling conventions and with the Language Environment-supplied Assembler prologue macro). If the caller is compiled with XPLINK, the behavior depends on the OSCALL suboption of the XPLINK compiler option. This suboption selects the behavior for linkage OS from among OS_DOWNSTACK, OS_UPSTACK, and OS_NOSTACK (the default). This means that applications, which use linkage OS to communicate among C or C++ functions, will need source changes when recompiled with XPLINK. See the description that follows for REFERENCE.
PLI	Designates an entry point (<i>identifier</i> in the syntax diagram) as a PL/I linkage entry point.
COBOL	Designates an entry point (<i>identifier</i> in the syntax diagram) as a COBOL linkage entry point.

#pragma

FORTTRAN	<p>Designates an entry point (<i>identifier</i> in the syntax diagram) as a FORTRAN linkage entry point.</p> <p>You can specify the RETURNCODE keyword with the FORTRAN keyword for C programs only. z/OS C/C++ does not support it for C++.</p> <p>RETURNCODE indicates to the compiler that the routine named by <i>identifier</i> is a FORTRAN routine, which returns an alternate return code. It also indicates that the routine is defined outside the compilation unit. You can retrieve the return code by using the <code>fortrc()</code> function. If the compiler finds the function definition inside the compilation unit, it issues an error message. Note that you can define functions outside the compilation unit, even if you do not specify the RETURNCODE keyword.</p>
OS_DOWNSTACK	<p>Designates an entry point (<i>identifier</i> in the syntax diagram) as an OS linkage entry point in XPLINK mode with a downward growing stack frame.</p> <p>If the function identified by <i>identifier</i> is defined within the compilation unit and is compiled using the NOXPLINK option, the compiler issues an error message.</p>
OS_UPSTACK	<p>Designates an entry point (<i>identifier</i> in the syntax diagram) as an OS linkage entry point in XPLINK mode with a traditional upward growing stack frame.</p> <p>This linkage is required for a new XPLINK downward-stack routine to be able to call a traditional upward-stack OS routine. This linkage explicitly invokes compatibility code to swap the stack between the calling and the called routines.</p> <p>If the function identified by <i>identifier</i> is defined within the compilation unit and is compiled using the XPLINK option, the compiler issues an error message. Typically, the <i>identifier</i> will not be defined in a compilation. This is acceptable. In this case, it is a reference to an external procedure that is separately compiled with NOXPLINK.</p>
OS_NOSTACK	<p>Designates an entry point (<i>identifier</i> in the syntax diagram) as an OS linkage entry point in XPLINK mode with no preallocated stack frame. An argument list is constructed containing the addresses of the actual arguments. The last item in this list has its high order bit set. Register 1 is set to point to this argument list. Register 13 points to a 72-byte save area that may not be followed by z/OS Language Environment control structures, such as the NAB. Register 14 contains the return address. Register 15 contains the entry point of the called function. This is synonymous with OS31_NOSTACK.</p>
OS31_NOSTACK	<p>Designates an entry point (<i>identifier</i> in the syntax diagram) as an OS linkage entry point in XPLINK mode with no preallocated stack frame.</p>
REFERENCE	<p>This is synonymous with OS_UPSTACK in non-XPLINK mode and synonymous with OS_DOWNSTACK in XPLINK mode. Unlike the linkage OS, this is not affected by the OSCALL suboption of XPLINK. Consider using this OS instead to make the source code portable between XPLINK and non-XPLINK.</p>

You can use a typedef in a #pragma linkage directive to associate a specific linkage convention with the typedef of a function.

```
typedef void func_t(void);
#pragma linkage (func_t,OS)
```

In the example, the #pragma linkage directive associates the OS linkage convention with the typedef func_t. This typedef can be used in C declarations wherever a function type specifies the type function of OS linkage type.

longname

z/OS The z/OS #pragma longname directive specifies that the compiler is to generate not-truncated and mixed case names in the object module that is produced by the compiler. These names can be up to 1024 characters in length.



If you use the #pragma longname directive for z/OS C and C++ programs, you must either use the binder to produce a program object in a PDSE, or you must use the prelinker. The binder, IPA Link step, and prelinker support the long name directory that is generated by the Object Library utility for autocall.

If you specify the NOLONGNAME compile option, the compiler ignores the #pragma longname directive. If you specify the LONGNAME compile option, the compiler ignores the #pragma noLongname.

Note: The z/OS C compiler defaults to the NOLONGNAME compile option, and the z/OS C++ compiler defaults to the LONGNAME compile option. Under z/OS C, if you specify the ALIAS compile option, the compiler creates a NAME control statement, but no ALIAS control statements. You can use the z/OS Object Library Utility to create a library of object modules with a long name directory which supports autocall of long name symbols.

If you have more than one preprocessor directive, #pragma longname may be preceded only by #pragma filetag, #pragma chars, #pragma langlvl, and #pragma target. Some directives, such as #pragma variable and #pragma linkage are sensitive to the name handling.

For z/OS C++, you must specify #pragma longname and #pragma noLongname before any code. Otherwise, the compiler issues a warning message.

If you use #pragma map to associate an external name with an identifier, the external name is produced in the object module. That is, #pragma map has the same behavior with or without the #pragma longname directive.

The #pragma noLongname directive directs the compiler to generate truncated and uppercase names in the object module produced by the compiler. When the #pragma noLongname directive is specified, only functions that do not have C++ linkage are given truncated and uppercase names. More details on external name mapping are provided in the section, “map” on page 242. Also, if you have more than one preprocessor directive, #pragma noLongname must be the first one.

#pragma

If you specify either `#pragma nolongname` or the `NOLONGNAME` option, and this results in mapping of two different source code names to the same object code name, the compiler will not issue an error message.

IPA Considerations for the #pragma longname

You must specify either the `LONGNAME` compile option or the `#pragma longname` preprocessor directive for the IPA Compile step (unless you are using the `c89` or `cc` utility from z/OS UNIX, both of which already specify the `LONGNAME` compiler option). Otherwise, you receive an unrecoverable compiler error.

map

z/OS The z/OS `#pragma map` directive tells the compiler to convert all references to an identifier to *"name"*.

`#pragma map` is a z/OS C/C++ directive and an addition to SAA standard. If you use the `#pragma map` directive, the C/C++ name in the source file is not visible in the object deck. The map name represents the object in the object deck.

#pragma map for z/OS C

For z/OS C, `#pragma map` has the form:

►► `#pragma map (—identifier—, —"name"—)` ◀◀

identifier A name of a data object or function with external linkage.

name The external name that the compiler binds to the given object or function. If the name is longer than 8 characters, you must use the binder and specify the `LONGNAME` compile option.

The directive can appear anywhere within a single compilation unit. It can appear before any declaration or definition of the named object or function.

You should enclose *name* in double quotation marks. The maximum length for external names is eight characters, unless the `LONGNAME` compile option is specified. The compiler keeps it as specified on the `#pragma map` directive in mixed case. It must not conflict with the name in another `#pragma map` or `#pragma csect` directive.

The map name is an external name, thus you must not use it in the source file to reference the object. If you use the map name in the source file to access the corresponding object, the compiler treats it as a new identifier.

The compiler produces an error message if you give more than one map name to an identifier. Two different identifiers can have the same map name.

The compiler resolves the identifiers appearing in the directive, including any type names used in the prototype argument list. The compiler resolves them as though the directive had appeared at file scope, independent of its actual point of occurrence.

For example:

```
extern "C" int func(int);
#pragma map(func, "funcnam1")    // maps ::func
```

#pragma map for z/OS C++

For z/OS C++, #pragma map has the form:

```

▶▶ #pragma map
▶▶ ( identifier
    | func_or_op_identifier (—argument_list—)
    , —"name" — )
▶▶

```

<i>identifier</i>	A name of a data object or a nonoverloaded function with external linkage.
<i>func_or_op_identifier</i>	A name of a function or operator with external linkage. The name can be qualified.
<i>argument_list</i>	A prototype list for the named function or operator.
<i>name</i>	The external name that is bound to the given object, function, or operator. If the name is longer than 8 characters you must use the binder.

The directive can appear anywhere within a single compilation unit. It can appear before any declaration or definition of the named object, function, or operator. The compiler resolves the identifiers appearing in the directive, including any type names used in the prototype argument list. It resolves them as though the directive had appeared at file scope, independent of its actual point of occurrence.

Note: The map name is not affected by the CONVLIT or the ASCII compiler options.

For example:

```

int func(int);

class X
{
public:
    void func(void);
#pragma map(func, "funcname1")    // maps ::func
#pragma map(X::func, "funcname2") // maps
X::func
};

```

In z/OS C++, you should not use #pragma map to map the following:

- z/OS C++ Member functions
- Overloaded functions
- Objects generated from templates
- Functions with z/OS C++ linkage, or linkage

Such mappings override the compiler-generated names, which could cause IPA Link or binder errors.

IPA Considerations for the #pragma map

The use of the #pragma map directive for variables will inhibit the global coalescing optimization of these variables during the IPA Link step.

margins

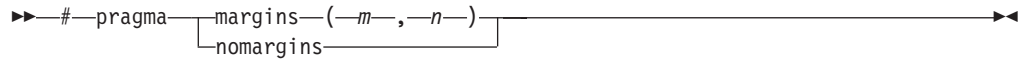
▶ z/OS The z/OS #pragma margins directive specifies the margins in the source file that are to be scanned for input to the compiler. You cannot specify columns (*m,n*)

#pragma

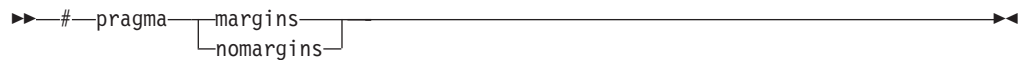
for z/OS C++. The `#pragma nomargins` directive specifies that the entire input source record is to be scanned for input to the compiler.

`#pragma margins` is a z/OS C/C++ directive and an addition to the SAA Standard.

#pragma margins for z/OS C



#pragma margins for z/OS C++



In the syntax diagram, you can specify the following parameters for z/OS C:

m The first column of the source input that contains a valid C program. The value of *m* must be greater than 0, and less than 32761.

Also, *m* must be less than or equal to the value of *n*.

n The last column of the source input that contains a valid C program. The value of *n* must be greater than 0, and less than 32761.

You can assign an asterisk (*) to *n*. The asterisk indicates the last column of the input record. For example, if you specify `#pragma margins(8,*)`, the compiler scans from column 8 to the end of the record for input source statements.

You can use `#pragma margins` and `#pragma` sequence together. If they reserve the same columns, `#pragma` sequence has priority and it reserves the columns for sequence numbers. For example, assume columns 1 to 20 are reserved for the margin, and columns 15 to 25 are reserved for sequence numbers. In this case, the margin will be from column 1 to 14, and the columns reserved for sequence numbers will be from 15 to 25.

The margin setting specified by the `#pragma margins` directive applies only to the source file or include file in which it is found. It has no effect on other `#include` files. The `#pragma margins` and the `#pragma nomargins` directives come into effect on the line following the directive. They remain in effect until the compiler encounters another `#pragma margins` or `#pragma nomargins` directive, or until the compiler reaches the end of the file.

If you use the compile options `MARGINS` or `NOMARGINS` with the `#pragma margins` or `#pragma nomargins` directives, the `#pragma` directives override the compile options. The compile option specified will be in effect up to, and including, the `#pragma margins` or `#pragma nomargins` directive.

For z/OS C++, the `#pragma margins` specifies that columns 1 through 72 in the input record are to be scanned for input to the compiler. The input file can have fixed or variable record length. The compiler ignores any text in the source input that does not fall within the range.

For z/OS C, the default setting is MARGINS(1,72) for fixed-length records, and NOMARGINS for variable-length records. For z/OS C++, the default is NOMARGINS.

namemangling (z/OS C++ only)

The `#pragma namemangling` directive sets the maximum length for external symbol names that are generated from C++ source code. Name mangling is the encoding of variable names into unique names so that linkers can separate common names in the language. With respect to the C++ language, name mangling is commonly used to facilitate the overloading feature and visibility within different scopes.

The syntax is shown in the diagram below. The pragma is unusual in that it is case-sensitive.



where:

ansi Indicates that the name mangling scheme complies with the C++ standard. The default value for *num_chars* is 64,000 characters, which is the maximum.

compat Indicates that the name mangling scheme is the same as that in earlier versions of the z/OS C/C++ and OS/390 C/C++ compiler, and is provided for compatibility with link modules created with earlier compilers. The default value for *num_chars* is 255 characters, which is the maximum.

num_chars
Represents a user-specified maximum for the length of a generated external symbol.

The pragma controls the final length of the mangled name if you specify a length. Although you are able to do this, the savings in storage space will be small. The default values are maximums.

noinline

z/OS The z/OS `#pragma noinline` directive is an addition to the SAA Standard.

The `#pragma noinline` specifies that the function is never inlined when you call it. This pragma has no effect when you specify NOAUTO with the z/OS C/C++ `INLINE` compiler option.

You can place the `#pragma noinline` directive anywhere in a C++ program. The directive must be at file scope in a C program.

The `#pragma noinline` directive is the only way to turn off inlining of functions that have been implicitly or explicitly inlined at compile time. It also takes precedence over the z/OS C++ `inline` keyword.

See “inline (z/OS C Only)” on page 234 for more information.

#pragma

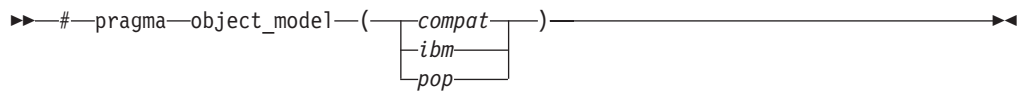
IPA Considerations for the #pragma noline

If you use either the `#pragma inline` or the `#pragma noline` directive in your source, you can later override them with an appropriate IPA Link control file directive during the IPA Link step. The compiler uses the IPA Link control file directive in the following cases:

- If you specify both the `#pragma noline` directive and the IPA Link control file `inline` directive for a function.
- If you specify both the `#pragma inline` directive and the IPA Link control file `noline` directive for a function.

object_model (z/OS C++ Only)

z/OS The `z/OS #pragma object_model` directive specifies the object model to use for the structures, unions, and classes that follow it. All classes in the same inheritance hierarchy must have the same object model. The following syntax diagram shows the object models that can be specified.



The arguments for the specifiable object models differ in the areas of layout for the virtual function table, support for virtual base classes, and name mangling scheme.

- `COMPAT` is compatible with name mangling and the virtual function table that was available with the previous releases of the C++ compiler.
- Use `ibm` if you want improved performance. Class hierarchies with many virtual base classes can benefit from this option because the size of the derived class is smaller and access to the virtual function table is faster.
- Use `pop` to restore the object model to that which was in effect prior to the last `#pragma object_model` statement, or to the default `object_model` if there were no previous `#pragma object_model` statements.

The following examples show the use of the `#pragma object_model(pop)` statement.

```
// Example 1
// Begin test1.C

/* default object model in effect */

#pragma object_model(ibm)
/* ibm object model in effect from this point */

#pragma object_model(pop)
/* restores the default object model */

// End test1.C

// Example 2
// Begin test2.C

/* default object model in effect */

#pragma object_model(ibm)
/* ibm object model in effect from this point */

#pragma object_model(compat)
/* compat object model is in effect */
```

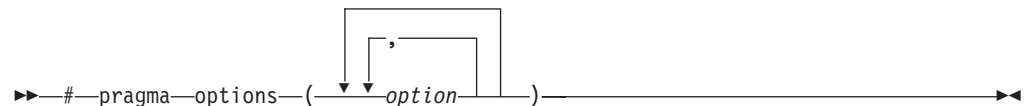


```
#pragma object_model(pop)
/* restores the ibm object model, which was in effect
   prior to the last #pragma object_model statement */

// End test2.C
```

options (z/OS C Only)

z/OS The z/OS #pragma options directive specifies a list of compile options that are to be processed as if you had typed them on the command line or on the CPARM parameter of the IBM- supplied cataloged procedures.



The only compile options that are allowed on a #pragma options directive are:

AGGREGATE NOAGGREGATE	ALIAS NOALIAS	ANSIALIAS NOANSIALIAS
ARCHITECTURE	CHECKOUT NOCHECKOUT	
GONUMBER NOGONUMBER	IGNERRNO NOIGNERRNO	INLINE NOINLINE
LIBANSI NOLIBANSI	MAXMEM NOMAXMEM	OBJECT NOOBJECT
OPTIMIZE NOOPTIMIZE	RENT NORENT	SERVICE NOSERVICE
SPILL NOSPILL	START NOSTART	TEST NOTEST
UPCONV NOUPCONV	TUNE NOTUNE	XREF NOXREF

If you use a compile option that contradicts the options that are specified on the #pragma options directive, the compile option overrides the options on the #pragma options directive.

If you specify an option more than once, the compiler uses the last one you specified.

IPA Considerations for the #pragma options

You cannot specify the IPA compile-time option for #pragma options.

option_override

z/OS With the z/OS #pragma option_override directive, you can specify optimization with more granularity in your applications. Specifically, this pragma defines subprogram (C function, C++ method) specific options that override those specified by the command line options when performing optimizations for code and data in that subprogram. This enables finer control of program optimization.

The subprogram-specific COMPACT option selects optimizations that tend to minimize code size in these functions while optimizing the rest of your application for execution speed.

The subprogram-specific OPTIMIZE option leaves specified functions unoptimized, while optimizing the rest of your application. The LEVEL option performs the same function as the OPTIMIZE option.

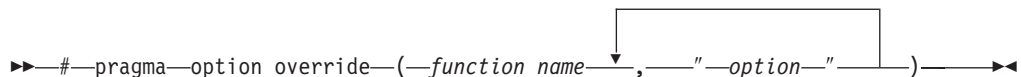
#pragma

One use of this directive is to isolate coding errors that occur only under optimization.

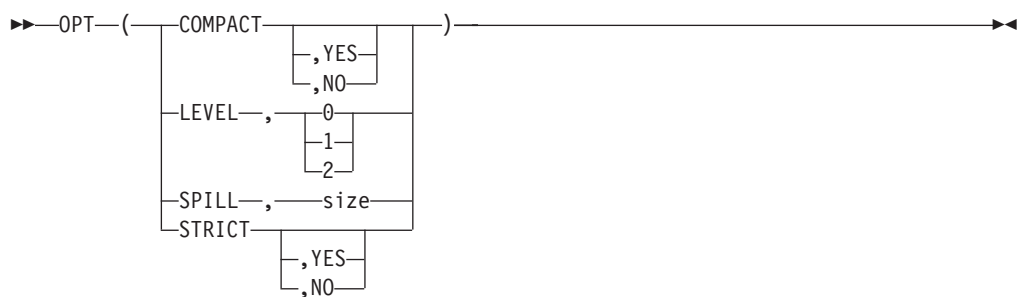
The subprogram-specific SPILL option allows you to specify large spill values for specific functions, while allowing the remaining subprograms to have smaller values.

The subprogram-specific STRICT option allows you to have rigidly controlled computational operations in some subprograms, while allowing faster operations where this control is not required.

In addition, you can also override the value of the STRICT option in order to change the rigor with which z/OS C/C++ performs numeric computations.



In the above syntax ² diagram, *option* can be one of the following:



In the above syntax diagram, *option* translates to the following equivalent z/OS C/C++ option settings:

option_override Value	Equivalent Option Setting
COMPACT	COMPACT
COMPACT, YES	COMPACT
COMPACT, NO	NOCOMPACT
LEVEL, 0	OPT(0)
LEVEL, 1	OPT(1)
LEVEL, 2	OPT(2)
SPILL, size	SPILL(size)

2. This pragma directive maintains compatibility with the AIX platform, even though the compiler options supported on the z/OS and AIX platforms are not the same. Thus, the option syntax for this directive is different from the z/OS C/C++ command line option syntax for the same option. Within this directive, options are grouped by category. OPT here signifies optimization-related options. Within the category are the actual options. The optimization-related options are LEVEL (equivalent to the command line option OPTIMIZE) and STRICT (equivalent to the command line option STRICT). This syntax attempts to be sufficiently platform-neutral such that this directive is portable between different platforms.

STRICT	STRICT
STRICT, YES	STRICT
STRICT, NO	NOSTRICT

To use the `LEVEL` suboption, you must specify a non- zero optimization level for your program, otherwise, the compiler ignores it. All other cases are meaningful, including increasing the optimization level.

The `#pragma option_override` directive only affects functions that are defined in the compile unit. The `pragma` directive can appear anywhere in the compilation unit. That is, it can appear before or after the function definition, before or after the function declaration, before or after the function has been referenced, and inside or outside the function definition.

Specifying this directive only affects the setting of the options you have specified. Notice that *option* can be enclosed in double quotation marks, so the options are not subject to macro expansion.

Following is an example of how you might use this directive. Suppose you compile the following code fragment containing the functions `foo()` and `faa()` using `OPT(1)`. Since it contains the `#pragma option_override(faa, "OPT(LEVEL, 0)")`, function `faa()` will not be optimized.

```
foo(){
    .
    .
    .
}

#pragma option_override(faa, "OPT(LEVEL, 0)")

faa(){
    .
    .
    .
}
```


IPA Considerations for the #pragma option_override

You cannot specify the IPA compile-time option for `#pragma option_override`.

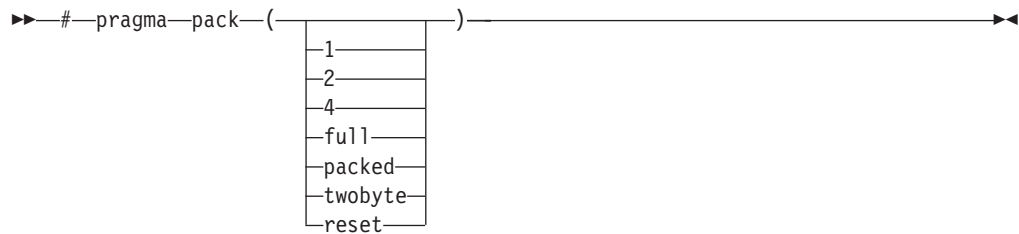
During IPA Compile processing, subprogram-specific options will be used to control IPA Compile-time optimizations.

During IPA Link processing, subprogram-specific options will be used to control IPA Link-time optimizations, as well as program partitioning. They will be retained, even if the related IPA Link command line option is specified.

pack

 The z/OS `#pragma pack` directive specifies the alignment rules to use for the structures, unions, and classes that follow it. The C compiler performs packing on *definitions* if you specify keyword `_Packed` and on *declarations* if you specify `#pragma pack`. The z/OS C++ compiler does not support keyword `_Packed`, so it can only perform packing on *declarations*. This means that the packing applies to type-specifiers and not declarators. Prior to using this `pragma` directive, you should understand the alignment rules for structures and for unions.

#pragma



where:

<code>full</code>	Is 4-byte boundary alignment. It is the system default boundary alignment. This is the same as <code>#pragma pack()</code> and <code>#pragma pack(4)</code> .
<code>packed</code>	Is 1-byte boundary alignment. This is the same as <code>#pragma pack(1)</code> .
<code>twobyte</code>	Is 2-byte boundary alignment. This is the same as <code>#pragma pack(2)</code> .
<code>reset</code>	Returns the alignment to the previous alignment rule.

The `#pragma pack` directive packs all structures and unions that follow it in the program along a boundary specified in the directive. It continues to pack until another `#pragma pack` directive changes the packing boundary. The `#pragma pack` directive does not apply to forward declarations of structures or unions. For example, in the following code fragment, the alignment for struct `S` is full. This is the rule when the declaration list is declared:

```
#pragma pack(packed)
struct S;
#pragma pack(full)
struct S { int i, j, k; };
```

The compiler packs declarations or types. This is different from the `_Packed` keyword in z/OS C, where packing is also performed on definitions. For portability, you should use `#pragma pack` instead of the `_Packed` keyword.

The `#pragma pack` directive does not have the same effect as declaring a structure as `_Packed`. The `_Packed` keyword removes all padding between structure members, while the `#pragma pack` directive only specifies the boundaries to align the members.

C++ If you are porting code from other platforms that contain `#pragma pack` directives or packed data, consider using the PORT compiler option to increase the syntax checking for the `#pragma pack` directive in your code. This option will allow you to adjust the error recovery action the compiler takes if the `#pragma pack` is incompatible with the z/OS C/C++ `#pragma pack`.

RELATED REFERENCES

- “Alignment of Unions” on page 64

page (z/OS C Only)

z/OS The z/OS `#pragma page` allows you to specify that a source listing begins at the top of a page. The parameter *pages* specifies the number of pages from the current page on which to begin writing the line of source code that follows the pragma.

```

>> #pragma page ( pages )

```

#pragma page() is the same as #pragma page(1): the source line that follows the pragma will start on a new page. If you write #pragma page(2), the listing will skip one blank page and the source line following the #pragma page will start on the second page after the current page. In all cases, the listing continues.

pagesize (z/OS C Only)

z/OS The z/OS #pragma pagesize directive sets the number of lines per page to *n* for the generated source listing.

```

>> #pragma pagesize ( n )

```

The default page size is 66 lines. The minimum page size that you should set is 25.

IPA Considerations for the #pragma pagesize

This #pragma has the same effect on the IPA Compile step as it does on a regular compilation. It has no effect on the IPA Link step.

priority (z/OS C++ Only)

z/OS The z/OS C++ #pragma priority directive specifies the order in which z/OS C++ runs constructors for static objects at run time. Destructors for these objects are run in reverse order during termination.

```

>> #pragma priority ( -n )

```

n is an integer literal in the range of INT_MIN to INT_MAX. The default value is 0. A negative value indicates a higher priority; a positive value indicates a lower priority.

z/OS C/C++ reserves the first 1024 priorities (INT_MIN to INT_MIN + 1023) for use by the compiler and its libraries. More than one #pragma priority can be specified within a compilation unit. The priority value specified in one pragma applies to the constructions of all global objects declared after this pragma and before the next one.

Note that the C++ Standard requires that all global objects within the same compilation unit be constructed from top to bottom, but it does not impose an ordering for objects declared in different compilation units. The #pragma priority tightens this up by imposing a construction order for all objects declared within the same load module. In order to be consistent with the Standard, there is a restriction that the priority values specified within the same compilation unit be strictly increasing.

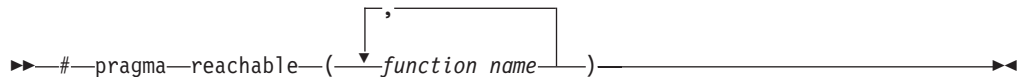
The effect of a #pragma priority exists only within one load module. A DLL is one load module. Therefore, #pragma priority cannot be used to control the

#pragma

construction order of objects in different DLLs. Please refer to *z/OS C/C++ Programming Guide* for further discussions on techniques used in handling DLL static object initialization.

reachable

> z/OS The z/OS #pragma reachable directive takes a function name, and declares that the point in the program after that function can be the target of a branch from some unknown location. That is, you can reach the instruction after the specified function from a point in your program other than the return statement in the named function.



When enabled, the reachable pragma provides information to the compiler that enables it to explore additional opportunities for optimization.

Unlike the leaves pragma, the reachable pragma is required by the compiler optimizer whenever the instruction following the call may receive control from some program point other than the return statement of the called function. If this condition is true and "reachable" is not specified, then the subprogram containing the call should not be compiled with OPT(1), OPT(2) or IPA. Also see "leaves" on page 238.

When you specify the LIBANSI compiler option, you tell the compiler that that you are using the system C run-time library, and not your own version of the library. In this case, the compiler checks whether the setjmp family of functions (setjmp, _setjmp, sigsetjmp, and _sigsetjmp) contain #pragma reachable. If the functions do not contain this pragma, the compiler assumes that the pragma is specified.

IPA Considerations for the #pragma reachable

If you specify the #pragma reachable directive in your source code in the IPA compile step, you cannot override the effects of this directive in the IPA link step.

Effect on IPA Link Step

If you specify the LIBANSI compile option for any compilation unit in the IPA compile step, the compiler generates information which indicates the setjmp() family of functions contain the reachable status. If you specify the NOLIBANSI option for the IPA link step, the attribute remains in effect.

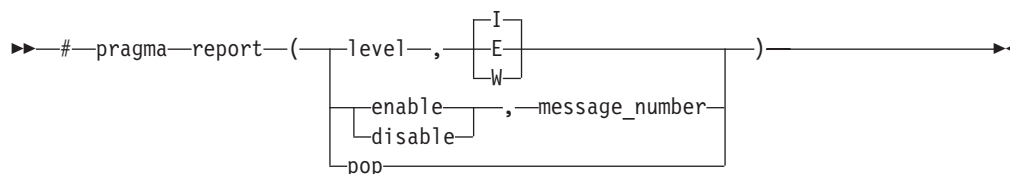
report (z/OS C++ only)

The #pragma report directive controls the generation of diagnostic messages by allowing you to specify a minimum severity level for a message in order for it to display, or by allowing you to enable or disable a specific message regardless of the prevailing report level.

The pragma takes precedence over #pragma info and most compiler options. For example, if you use #pragma report to disable a compiler message, that message will not be displayed with any FLAG compiler option setting. Similarly, if you specify the SUPPRESS compiler option for a message but also specify #pragma report(enable) for the same message, the pragma will prevail.

#pragma

The #pragma report(pop) reverts the report level to that which was previously in effect. If no previous report level has been specified, a warning is issued, and the report level remains unchanged. The default report level is Informational (I), which displays messages of all types.



where:

level Specifies a minimum severity level for diagnostic messages in order for them to display.

E Indicates that only error messages will display.

W Indicates that warning and error messages will display.

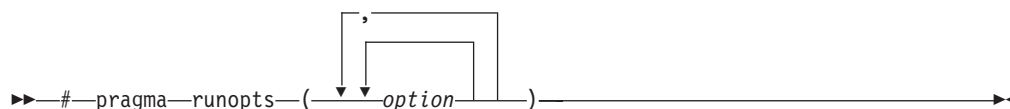
I Indicates that all diagnostic messages will display. Informational messages are of the lowest severity.

message_number

Represents a message identifier, which consists of a prefix followed by the message number, for example, CCN1004.

runopts

z/OS The z/OS #pragma runopts directive specifies a list of run-time options that z/OS C/C++ uses at execution time.



Specify your #pragma runopts directive in the compilation unit that contains main. If more than one compilation unit contains a #pragma runopts directive, unpredictable results can occur. The #pragma runopts directive only affects compilation units containing main().

If a suboption to #pragma runopts is not a valid C or C++ token, you can surround the suboptions to #pragma runopts in double quotes. For example, use:

```
#pragma runopts ( " RPTSTG(ON) TEST(,,VADTCPIP&1.2.3.4:*) " )
```

instead of:

```
#pragma runopts ( RPTSTG(ON) TEST(,,VADTCPIP&1.2.4.3:*) )
```

Refer to “target (z/OS C Only)” on page 256 for information about how #pragma target and the TARGET compile-time option affect #pragma runopts.

IPA Considerations for the #pragma runopts

This #pragma only affects the IPA Compile step if you specify the OBJECT suboption of the IPA compiler option.

#pragma

The IPA Compile step passes the effects of this directive to the IPA Link step.

Consider if you specify `ARGPARSE|NOARGPARSE`, `EXECOPS|NOEXECOPS`, `PLIST`, or `REDIR|NOREDİR` either on the `#pragma runopts` directive or as a compile-time option on the IPA Compile step, and then specify the compile-time option on the IPA Link step. In this case, you override the value that you specified on the IPA Compile step.

If you specify the `TARGET` compile-time option on the IPA Link step, it has the following effects on `#pragma runopts` :

- It overrides the value you specified for `#pragma runopts(ENV)`. If you specify `TARGET(LE)` or `TARGET()`, the compiler sets the value of `#pragma runopts(ENV)` to `MVS`. If you specify `TARGET(IMS)`, the compiler sets the value of `#pragma runopts(ENV)` to `IMS`.
- It may override the value you specified for `#pragma runopts(PLIST)`. If you specify `TARGET(LE)` or `TARGET()`, and you specified something other than `HOST` for `#pragma runopts(PLIST)`, the compiler sets the value of `#pragma runopts(PLIST)` to `HOST`. If you specify `TARGET(IMS)`, the compiler sets the value of `#pragma runopts(PLIST)` to `IMS`.

For `#pragma runopts` options other than those that are listed above, the IPA Link step follows these steps to determine which `#pragma runopts` value to use:

1. The IPA Link step uses the `#pragma runopts` specification from the `main()` routine, if the routine exists.
2. If no `main()` routine exists, the IPA Link step follows these steps:
 - a. If you define the `CEEUOPT` variable, the IPA Link step uses the `#pragma runopts` value from the first compilation unit that it finds that contains `CEEUOPT`.
 - b. If you have not defined the `CEEUOPT` variable in any compilation unit, the IPA Link step uses the `#pragma runopts` value from the first compilation unit that it processes.

The sequence of compilation unit processing is arbitrary.

To avoid problems, you should specify `#pragma runopts` only in your `main()` routine. If you do not have a `main()` routine, specify it in only one other module.

sequence

z/OS The `z/OS #pragma sequence` directive specifies the section of the input record that is to contain sequence numbers. The `#pragma nosequence` directive specifies that the input record does not contain sequence numbers.

`#pragma sequence` is specific to `z/OS C/C++` and an addition to the `SAA Standard`.

#pragma sequence for z/OS C

►► `#pragma` — `sequence` — `(—m—, —n—)` —————►
 └ `nosequence` —————►

#pragma sequence for z/OS C++

In the syntax diagram you can specify the following parameters for z/OS C:

m The column number of the left-hand margin. The value of *m* must be greater than 0, and less than 32761.

Also, *m* must be less than or equal to the value of *n*.

n The column number of the right-hand margin. The value of *n* must be greater than 0, and less than 32761.

You can assign an asterisk (*) to *n* that indicates the last column of the input record. For example, SEQUENCE(74,*) indicates that sequence numbers are between column 74 and the end of the input record.

You can use #pragma sequence and #pragma margins together. If they reserve the same columns, #pragma sequence has priority, and z/OS C/C++ reserves the columns for sequence numbers. For example, consider if the columns reserved for the margin are 1 to 20 and the columns reserved for sequence numbers are 15 to 25. In this case, the margin will be from column 1 to 14, and the columns reserved for sequence numbers will be from 15 to 25. For more information on the #pragma margins directive, refer to “margins” on page 243.

The sequence setting specified by the #pragma sequence directive applies only to the file (source file or include file) that contains it. The setting has no effect on other #include files in the file. The sequence number area specified on the #pragma sequence directive comes into effect on the line following the directive. It remains in effect until it encounters another #pragma sequence or a #pragma nosequence directive or until it reaches the end of the file.

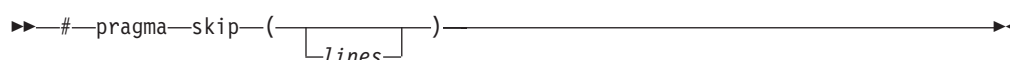
If you use the compile-time options SEQUENCE|NOSEQUENCE with the #pragma sequence or #pragma nosequence directives, the #pragma directive overrides the compile options. The compile option is in effect up to, and including, the #pragma sequence or the #pragma nosequence directive.

For z/OS C++, the #pragma sequence directive defines that columns 73 through 80 of the input record (fixed or variable length) contain sequence numbers. You cannot specify columns (*m*,*n*). The default compile option for z/OS C++ is NOSEQUENCE.

For z/OS C, the default setting is SEQUENCE(73,80) for fixed-length records, and NOSEQUENCE for variable length records.

skip (z/OS C Only)

z/OS The z/OS #pragma skip directive skips the specified number of lines in the generated source listing. The value of *lines* must be a positive integer less than 255. If you omit *lines*, the compiler skips one line.



#pragma strings

z/OS The z/OS #pragma strings directive sets the storage type for strings. It specifies that the compiler can place strings into read-only memory or must place strings into read/write memory.

►► #pragma strings (

writable
writeable
readonly

) ►►

This pragma must appear before any z/OS C or C++ code in a file.

C and C++ strings are read-only by default because the ROSTRING compiler option, which informs the compiler that string literals are read-only, is now the compiler default. Formerly, you could use the ROSTRING compiler option instead of using the #pragma strings(readonly) directive.

IPA Considerations for the #pragma strings

During the IPA Link step, the compiler compares the #pragma strings specifications for individual compilation units. If it finds differences, it treats the strings as if you specified #pragma strings(writeable) for all compilation units.

subtitle (z/OS C Only)

z/OS The z/OS #pragma subtitle directive places the text that is specified by *subtitle* on all subsequent pages of the generated source listing.

►► #pragma subtitle (—" *subtitle* "—) ►►

target (z/OS C Only)

z/OS The z/OS #pragma target directive specifies the run-time environment for which z/OS C/C++ creates the object.

Note: You cannot specify the release suboptions using the #pragma target directive as you can with the TARGET compile option.

►► #pragma target (

LE
IMS

) ►►

The compiler generates code to run under these options:

LE Generates code to run under the z/OS Language Environment run-time library. This is the default behavior, which acts the same as specifying #pragma target().

IMS Generates object code to run under IMS.

#pragma

If you have more than one preprocessor directive, the only #pragma directives that can precede #pragma target are #pragma filetag, #pragma chars, #pragma langlvl, and #pragma longname.

Specifying #pragma target() or #pragma target(LE) has the following effects on #pragma runopts(ENV) and #pragma runopts(PLIST):

- If you did not specify values for #pragma runopts(ENV) or #pragma runopts(PLIST), the compiler sets the #pragma s to #pragma runopts(ENV(MVS)) and #pragma runopts(PLIST(HOST)).
- If you did specify values for #pragma runopts(ENV) or #pragma runopts(PLIST), the values do not change.

Specifying #pragma target(IMS) has the following effects on #pragma runopts(ENV) and #pragma runopts(PLIST) :

- If you did not specify values for #pragma runopts(ENV) or #pragma runopts(PLIST), the compiler sets the #pragma s to #pragma runopts(ENV(IMS)) and #pragma runopts(PLIST(OS)).
- If you did specify values for #pragma runopts(ENV) or #pragma runopts(PLIST), the values do not change.

IPA Considerations for the #pragma target

This #pragma only affects the IPA Compile step if you specify the OBJECT suboption of the IPA compiler option.

The IPA Compile step passes the effects of this #pragma directive to the IPA Link step.

If you specify different #pragma target directives for different compilation units, the IPA Link step uses the ENV and PLIST information from the compilation unit containing main(). If there is no main(), it uses information from the first compilation unit it finds. If you specify the TARGET compile option for the IPA Link step, it overrules the #pragma target directive.

title (z/OS C Only)

z/OS The z/OS #pragma title directive places the text that is specified by *title* on all subsequent pages of the generated source listing.

```
▶▶#pragma title ("title")▶▶
```

variable

z/OS The z/OS #pragma variable directive specifies that z/OS C/C++ is to use the named external object in either a reentrant or non-reentrant fashion. If an object is marked as RENT, its references or its definition will be in the writable static area that is in modifiable storage. If an object is marked as NORENT, its references or definition is in the code area and is in potentially read-only storage.

```
▶▶#pragma variable (identifier, 

|        |
|--------|
| RENT   |
| NORENT |

)▶▶
```

#pragma

NORENT does not apply to, and has no affect on, program variables with static storage class. z/OS C/C++ always includes these variables with the writable static variables. Variables are reentrant by default for C++ so that RENT has no affect.

The #pragma variable directive is an addition to the SAA Standard.

You can use the ROCONST and RENT compiler options so that const variables are not placed into the Writeable Static Area instead of using the #pragma variable(var_name, NORENT) directive.

If the specification for a const variable in a #pragma variable directive conflicts with the ROCONST option, the pragma directive takes precedence over the compiler option, and the compiler issues an informational message.

The following restrictions apply:

- If an identifier is defined in one compilation unit and used in another, the reentrant or non-reentrant status of the variable must be the same in all compilation units.
- A non-reentrant pointer variable cannot take an address as an initializer: the compiler will treat the variable as reentrant if necessary (in other words, it will ignore the pragma). Initializers for non-reentrant variables should be compile-time constants. Due to code relocation during execution time, an address in a program that has both reentrant and non-reentrant variables is never considered a compile-time constant. This restriction includes the addresses of string literals.

The following code fragment leads to undefined behavior when compiled with the RENT option.

```
int i;  
int *p = &i;  
#pragma variable(p norent)
```

The variable i is reentrant, but the pointer p is non-reentrant. If the code is in a DLL, there will only be one copy of p but multiple copies of i, one for each caller of the DLL.

sizeof

z/OS The z/OS #pragma sizeof directive toggles the behavior of the sizeof operator between that of the C and C++ compilers prior to and including the C/C++ MVS/ESA Version 3 Release 1 product, and the z/OS C/C++ feature. As explained below, the difference occurs only when using sizeof on function return types. Other behaviors of sizeof remain the same.

Specify the pragma as follows:

```
➤ #pragma sizeof ( ON ) ➤  
                  RESUME
```

When using the sizeof operator, the z/OS C and C++ compilers prior to and including C/C++ MVS/ESA Version 3 Release 1, returned the size of the widened type instead of the original type for function return types. For example, in the following code fragment, using the older compilers, i has a value of 4.

```
char foo();  
i = sizeof foo();
```

Using the z/OS C/C++ compiler, `i` has a the value of 1, which is the size of the original type, `char`.

After a `#pragma wsizeof(on)` is encountered in a source program, all subsequent `sizeof` operators return the widened size for function return types. The behavior prior to the `#pragma wsizeof(on)`, which can be the old or current behavior, is saved. z/OS C/C++ reinstates this saved behavior when it encounters a matching `#pragma wsizeof(resume)`. The saving action works on a stack. That is, a *resume* reinstates the most recently saved state as the following example demonstrates:

```
/* Normal behavior of sizeof to start with.          */
/* ... some code here ...                            */

#pragma wsizeof(on) /* (1) old behavior of sizeof    */
...
#pragma wsizeof(on) /* (2) old behavior of sizeof    */
...
#pragma wsizeof(resume) /* matches (2)              */
/* still old behavior of sizeof                      */
...
#pragma wsizeof(resume) /* matches (1)              */
/* normal behavior of sizeof                         */
```

The compiler will match *on* and *resume* throughout the entire compile unit. That is, the effect of a `#pragma wsizeof(on)` can extend beyond a header file. Ensure the *on* and *resume* pragmas are matched in your compile unit.

Note: Dangling the *resume* pragma leads to undefined behavior. The effect of an unmatched *on* pragma can extend to the end of the source file.

Use the `wsizeof` pragma in old header files, where you require the old behavior of the `sizeof` operator. By guarding the header file with a `#pragma wsizeof(on)` at the start of the header, and a `#pragma wsizeof(resume)` at the end, you can use the old header file with new applications.

Using the `WSIZEOF` compile option and `#pragma wsizeof`

The z/OS `WSIZEOF` compile option has *exactly* the same effect as inserting a `#pragma wsizeof(on)` at the beginning of the source file. If another `#pragma wsizeof` exists in the source code, z/OS C/C++ toggles the behavior of the `sizeof` operator, as described above.

You can use the `WSIZEOF` compile option to save editing your source when you want the old behavior of the `sizeof` operator for your entire source file.

IPA Considerations for the z/OS `#pragma wsizeof`

During the IPA Compile step, the size of each function return value is resolved during source processing. The IPA Compile and Link steps do not alter these sizes. The IPA object code from compilation units with different `wsizeof` settings is merged together during the IPA Link step.

#pragma

Chapter 10. Namespaces

► **C++** A *namespace* is an optionally named scope. You declare names inside a namespace as you would for a class or an enumeration. You can access names declared inside a namespace the same way you access a nested class name by using the scope resolution (::) operator. However namespaces do not have the additional features of classes or enumerations. The primary purpose of the namespace is to add an additional identifier (the name of the namespace) to a name.

RELATED REFERENCES

- “C++ Scope Resolution Operator ::” on page 102

Defining Namespaces

► **C++** In order to uniquely identify a namespace, use the **namespace** keyword.

Syntax — namespace

► namespace { —namespace_body— } ►

identifier

The *identifier* in an original namespace definition is the name of the namespace. The identifier may not be previously defined in the declarative region in which the original namespace definition appears, except in the case of extending namespace. If an identifier is not used, the namespace is an *unnamed namespace*.

RELATED REFERENCES

- “Unnamed Namespaces” on page 264

Declaring Namespaces

► **C++** The identifier used for a namespace name should be unique. It should not be used previously as a global identifier.

```
namespace Raymond {  
    // namespace body here...  
}
```

In this example, Raymond is the identifier of the namespace. If you intend to access a namespace’s elements, the namespace’s identifier must be known in all translation units.

RELATED REFERENCES

- “Global Scope” on page 2

Creating a Namespace Alias

► **C++** An alternate name can be used in order to refer to a specific namespace identifier.

```
namespace INTERNATIONAL_BUSINESS_MACHINES {
    void f();
}

namespace IBM = INTERNATIONAL_BUSINESS_MACHINES;
```

In this example, the IBM identifier is an alias for INTERNATIONAL_BUSINESS_MACHINES. This is useful for referring to long namespace identifiers.

If a namespace name or alias is declared as the name of any other entity in the same declarative region, a compiler error will result. Also, if a namespace name defined at global scope is declared as the name of any other entity in any global scope of the program, a compiler error will result.

RELATED REFERENCES

- “Global Scope” on page 2

Creating an Alias for a Nested Namespace

C++ Namespace definitions hold declarations. Since a namespace definition is a declaration itself, namespace definitions can be nested.

An alias can also be applied to a nested namespace.

```
namespace INTERNATIONAL_BUSINESS_MACHINES {
    int j;
    namespace NESTED_IBM_PRODUCT {
        void a() { j++; }
        int j;
        void b() { j++; }
    }
}

namespace NIBM = INTERNATIONAL_BUSINESS_MACHINES::NESTED_IBM_PRODUCT
```

In this example, the NIBM identifier is an alias for the namespace NESTED_IBM_PRODUCT. This namespace is nested within the INTERNATIONAL_BUSINESS_MACHINES namespace.

Extending Namespaces

C++ Namespaces are extensible. You can add subsequent declarations to a previously defined namespace. Extensions may appear in files separate from or attached to the original namespace definition. For example:

```
namespace X { // namespace definition
    int a;
    int b;
}

namespace X { // namespace extension
    int c;
    int d;
}

namespace Y { // equivalent to namespace X
    int a;
    int b;
    int c;
    int d;
}
```


In this example, namespace X is defined with a and b and later extended with c and d. namespace X now contains all four members. You may also declare all of the required members within one namespace. This method is represented by namespace Y. This namespace contains a, b, c, and d.

Namespaces and Overloading

C++ You can overload functions across namespaces. For example:

```
// Original X.h:
f(int);

// Original Y.h:
f(char);

// Original program.c:
#include "X.h"
#include "Y.h"

void z()
{
    f('a'); // calls f(char) from Y.h
}
```

Namespaces can be introduced to the previous example without drastically changing the source code.

```
// New X.h:
namespace X {
    f(int);
}

// New Y.h:
namespace Y {
    f(char);
}

// New program.c:
#include "X.h"
#include "Y.h"

using namespace X;
using namespace Y;

void z()
{
    f('a'); // calls f() from Y.h
}
```

In program.c, function void z() calls function f(), which is a member of namespace Y. If you place the using directives in the header files, the source code for program.c remains unchanged.

RELATED REFERENCES

- “Chapter 11. Overloading” on page 269

Unnamed Namespaces

► C++ A namespace with no identifier before an opening brace produces an *unnamed namespace*. Each translation unit may contain its own unique unnamed namespace. The following example demonstrates how unnamed namespaces are useful.

```
#include <iostream>

using namespace std;

namespace {
    const int i = 4;
    int variable;
}

int main()
{
    cout << i << endl;
    variable = 100;
    return 0;
}
```

In the previous example, the unnamed namespace permits access to `i` and `variable` without using a scope resolution operator.

The following example illustrates an improper use of unnamed namespaces.

```
#include <iostream>

using namespace std;

namespace {
    const int i = 4;
}

int i = 2;

int main()
{
    cout << i << endl; // error
    return 0;
}
```

Inside `main`, `i` causes an error because the compiler cannot distinguish between the global name and the unnamed member with the same name. In order for the previous example to work, the namespace must be uniquely identified with an identifier and `i` must specify the namespace it is using.

You can extend an unnamed namespace within the same translation unit. For example:

```
#include <iostream>

using namespace std;

namespace {
    int variable;
    void funct (int);
}

namespace {
    void funct (int i) { cout << i << endl; }
}
```

```
int main()
{
    funct(variable);
    return 0;
}
```

both the prototype and definition for `funct` are members of the same unnamed namespace.

Note: Items defined in an unnamed namespace have internal linkage. Rather than using the keyword **static** to define items with internal linkage, define them in an unnamed namespace instead.

RELATED REFERENCES

- “Program Linkage” on page 5
- “Internal Linkage” on page 5

Namespace Member Definitions

C++ A namespace can define its own members within itself or externally using explicit qualification. The following is an example of a namespace defining a member internally:

```
namespace A {
    void b() { /* definition */ }
}
```

Within namespace `A` member `void b()` is defined internally.

A namespace can also define its members externally using explicit qualification on the name being defined. The entity being defined must already be declared in the namespace and the definition must appear after the point of declaration in a namespace that encloses the declaration’s namespace.

The following is an example of a namespace defining a member externally:

```
namespace A {
    namespace B {
        void f();
    }
    void B::f() { /* defined outside of B */ }
}
```

In this example, function `f()` is declared within namespace `B` and defined (outside `B`) in `A`.

Namespaces and Friends

C++ Every name first declared in a namespace is a member of that namespace. If a friend declaration in a non-local class first declares a class or function, the friend class or function is a member of the innermost enclosing namespace.

The following is an example of this structure:

```
// f has not yet been defined
void z(int);
namespace A {
    class X {
```

```

        friend void f(X); // A::f is a friend
    };
    // A::f is not visible here
    X x;
    void f(X) { /* definition */ } // f() is defined and known to be a friend
}

using A::x;

void z()
{
    A::f(x); // OK
    A::X::f(x); // error: f is not a member of A::X
}

```

In this example, function `f()` can only be called through namespace `A` using the call `A::f(s)`. Attempting to call function `f()` through class `X` using the `A::X::f(x)`; call results in a compiler error. Since the friend declaration first occurs in a non-local class, the friend function is a member of the innermost enclosing namespace and may only be accessed through that namespace.

RELATED REFERENCES

- “Friends” on page 310

Using Directive

C++ A *using directive* provides access to all namespace qualifiers and the scope operator. This is accomplished by applying the `using` keyword to a namespace identifier.

Syntax — Using directive

► `using namespace name;` ◀

The *name* must be a previously defined namespace. The using directive may be applied at the global and local scope but not the class scope. Local scope takes precedence over global scope by hiding similar declarations.

If a scope contains a using directive that nominates a second namespace and that second namespace contains another using directive, the using directive from the second namespace will act as if it resides within the first scope.

```

namespace A {
    int i;
}
namespace B {
    int i;
    using namespace A;
}
void f()
{
    using namespace B;
    i = 7; // error
}

```

In this example, attempting to initialize `i` within function `f()` causes a compiler error, because function `f()` cannot know which `i` to call; `i` from namespace `A`, or `i` from namespace `B`.

RELATED REFERENCES

- “The using Declaration and Class Members” on page 322

The using Declaration and Namespaces

C++ A *using declaration* provides access to a specific namespace member. This is accomplished by applying the using keyword to a namespace name with its corresponding namespace member.

Syntax — Using declaration

►—using—namespace—::—member—◄◄

In this syntax diagram, the qualifier name follows the using declaration and the *member* follows the qualifier name. For the declaration to work, the member must be declared inside the given namespace. For example:

```
namespace A {  
    int i;  
    int k;  
    void f;  
    void g;  
}
```

```
using A::k
```

In this example, the using declaration is followed by A, the name of namespace A, which is then followed by the scope operator (::), and k. This format allows k to be accessed outside of namespace A through a using declaration. After issuing a using declaration, any extension made to that specific namespace will not be known at the point at which the using declaration occurs.

In the incremental compiler, all names in all extents of a namespace will be made visible by a using declaration regardless of their positions.

Overloaded versions of a given function must be included in the namespace prior to that given function’s declaration. A using declaration may appear at namespace, block and class scope.

RELATED REFERENCES

- “The using Declaration and Class Members” on page 322

Explicit Access

C++ To explicitly qualify a member of a namespace, use the namespace identifier with a :: scope resolution operator.

Syntax — Explicit access qualification

►—namespace_name—::—member—◄◄

For example:

```
namespace VENDITTI {  
    void j()  
};  
  
VENDITTI::j();
```

In this example, the scope resolution operator provides access to the function `j` held within namespace `VENDITTI`. The scope resolution operator `::` is used to access identifiers in both global and local namespaces. Any identifier in an application can be accessed with sufficient qualification. Explicit access cannot be applied to an unnamed namespace.

RELATED REFERENCES

- “C++ Scope Resolution Operator `::`” on page 102

Chapter 11. Overloading

C++ If you specify more than one definition for a function name or an operator in the same scope, you have *overloaded* that function name or operator.

An *overloaded declaration* is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different types.

If you call an overloaded function name or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called *overload resolution*.

RELATED REFERENCES

- “Overloading Functions”
- “Overloading Operators” on page 271
- “Overload Resolution” on page 280

Overloading Functions

C++ You overload a function name *f* by declaring more than one function with the name *f* in the same scope. The declarations of *f* must differ from each other by the types and/or the number of arguments in the argument list. When you call a overloaded function named *f*, the correct function is selected by comparing the argument list of the function call with the parameter list of each of the overloaded candidate functions with the name *f*. A *candidate function* is a function that can be called based on the context of the call of the overloaded function name.

Consider a function `print`, which displays an **int**. As shown in the following example, you can overload the function `print` to display other types, for example, **double** and **char***. You can have three functions with the same name, each performing a similar operation on a different data type:

```
#include <iostream>
using namespace std;

void print(int i) {
    cout << " Here is int " << i << endl;
}
void print(double f) {
    cout << " Here is float " << f << endl;
}

void print(char* c) {
    cout << " Here is char* " << c << endl;
}

int main() {
    print(10);
    print(10.10);
    print("ten");
}
```

The following is the output of the above example:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

Restrictions on Overloaded Functions

► C++ You cannot overload the following function declarations if they appear in the same scope. Note that this list applies only to explicitly declared functions and those that have been introduced through **using** declarations:

- Function declarations that differ only by return type. For example, you cannot declare the following declarations:

```
int f();
float f();
```

- Member function declarations that have the same name and the same parameter types, but one of these declarations is a static member function declaration. For example, you cannot declare the following two member function declarations of `f()`:

```
struct A {
    static int f();
    int f();
};
```

- Member function template declarations that have the same name, the same parameter types, and the same template parameter lists, but one of these declarations is a static template member function declaration.
- Function declarations that have equivalent parameter declarations. These declarations are not allowed because they would be declaring the same function.
- Function declarations with parameters that differ only by the use of **typedef** names that represent the same type. Note that a **typedef** is a synonym for another type, not a separate type. For example, the following two declarations of `f()` are declarations of the same function:

```
typedef int I;
void f(float, int);
void f(float I);
```

- Function declarations with parameters that differ only because one is a pointer and the other is an array. For example, the following are declarations of the same function:

```
f(char*);
f(char[10]);
```

The first array dimension is insignificant when differentiating parameters; all other array dimensions are significant. For example, the following are declarations of the same function:

```
g(char(*)[20]);
g(char[5][20]);
```

The following two declarations are *not* equivalent:

```
g(char(*)[20]);
g(char(*)[40]);
```

- Function declarations with parameters that differ only because one is a function type and the other is a pointer to a function of the same type. For example, the following are declarations of the same function:

```
void f(int(float));
void f(int (*)(float));
```

- Function declarations with parameters that differ only because of **const** and **volatile** qualifiers. This only applies if you apply any of these qualifiers appear at the outermost level of an parameter type specification. For example, the following are declarations of the same function:


```
int f(int);
int f(const int);
int f(volatile int);
```

Note that you can differentiate parameters with **const** and **volatile** qualifiers if you apply these qualifiers *within* a parameter type specification. For example, the following declarations are *not* equivalent:

```
void g(int*);
void g(const int*);
void g(volatile int*);
```

The following declarations are also not equivalent:

```
void g(float&);
void g(const float&);
void g(volatile float&);
```

- Function declarations with parameters that differ only because their default arguments differ. For example, the following are declarations of the same function:

```
void f(int);
void f(int i = 10);
```

- Multiple functions with extern "C" language-linkage and the same name, regardless of whether their parameter lists are different.

RELATED REFERENCES

- “The using Declaration and Namespaces” on page 267
- “typedef” on page 43
- “volatile and const Qualifiers” on page 69
- “Linkage Specifications — Linking to Non-C++ Programs” on page 7

Overloading Operators

> C++ You can redefine or overload the function of most built-in operators in C++. These operators can be overloaded globally or on a class-by-class basis. Overloaded operators are implemented as functions and can be member functions or global functions.

An overloaded operator is called an *operator function*. You declare an operator function with the keyword **operator** preceding the operator. Overloaded operators are distinct from overloaded functions, but like overloaded functions, they are distinguished by the number and types of operands used with the operator.

Consider the standard + (plus) operator. When this operator is used with operands of different standard types, the operators have slightly different meanings. For example, the addition of two integers is not implemented in the same way as the addition of two floating-point numbers. C++ allows you to define your own meanings for the standard C++ operators when they are applied to class types. In the following example, a class called `complex` is defined to model complex numbers, and the + (plus) operator is redefined in this class to add two complex numbers.

CCNX12B

// This example illustrates overloading the plus (+) operator.

```
#include <iostream>
using namespace std;

class complex
{
```

```

        double real,
            imag;
public:
    complx( double real = 0., double imag = 0.); // constructor
    complx operator+(const complx&) const;      // operator+()
};

// define constructor
complx::complx( double r, double i )
{
    real = r; imag = i;
}

// define overloaded + (plus) operator
complx complx::operator+ (const complx& c) const
{
    complx result;
    result.real = (this->real + c.real);
    result.imag = (this->imag + c.imag);
    return result;
}

int main()
{
    complx x(4,4);
    complx y(6,6);
    complx z = x + y; // calls complx::operator+()
}

```

You can overload any of the following operators:

+	-	*	/	%	^	&		~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	<<=	>>=	==	!=
<=	>=	&&		++	--	,	->*	->
()	[]	new	delete	new[]	delete[]			

where () is the function call operator and [] is the subscript operator.

You can overload both the unary and binary forms of the following operators:

+ **-** ***** **&**

You cannot overload the following operators:

. **.*** **::** **?:**

You cannot overload the preprocessor symbols **#** and **##**.

An operator function can be either a nonstatic member function, or a nonmember function with at least one parameter that has class, reference to class, enumeration, or reference to enumeration type.

You cannot change the precedence, grouping, or the number of operands of an operator.

An overloaded operator (except for the function call operator) cannot have default arguments or an ellipsis in the argument list.

You must declare the overloaded **=**, **[]**, **()**, and **->** operators as nonstatic member functions to ensure that they receive lvalues as their first operands.

The operators **new**, **delete**, **new[]**, and **delete[]** do not follow the general rules described in this section.

All operators except the **=** operator are inherited.

RELATED REFERENCES

- “Free Store” on page 353

Overloading Unary Operators

C++ You overload a unary operator with either a nonstatic member function that has no parameters, or a nonmember function that has one parameter. Suppose a unary operator **@** is called with the statement **@t**, where **t** is an object of type **T**. A nonstatic member function that overloads this operator would have the following form:

```
return_type operator@()
```

A nonmember function that overloads the same operator would have the following form:

```
return_type operator@(T)
```

An overloaded unary operator may return any type.

The following example overloads the **!** operator:

```
#include <iostream>
using namespace std;

struct X { };

void operator!(X) {
    cout << "void operator!(X)" << endl;
}

struct Y {
    void operator!() {
        cout << "void Y::operator!()" << endl;
    }
};

struct Z { };

int main() {
    X ox; Y oy; Z oz;
    !ox;
    !oy;
    // !oz;
}
```

The following is the output of the above example:

```
void operator!(X)
void Y::operator!()
```

The operator function call **!ox** is interpreted as **operator!(x)**. The call **!oy** is interpreted as **y.operator!()**. (The compiler would not allow **!oz** because the **!** operator has not been defined for class **Z**.)

RELATED REFERENCES

- “Unary Expressions” on page 113

Overloading Binary Operators

C++ You overload a binary unary operator with either a nonstatic member function that has one parameter, or a nonmember function that has two parameters. Suppose a binary operator @ is called with the statement `t @ u`, where `t` is an object of type `T`, and `u` is an object of type `U`. A nonstatic member function that overloads this operator would have the following form:

```
return_type operator@(T)
```

A nonmember function that overloads the same operator would have the following form:

```
return_type operator@(T, U)
```

An overloaded binary operator may return any type.

The following example overloads the `*` operator:

```
struct X {  
  
    // member binary operator  
    void operator*(int) { }  
};  
  
// non-member binary operator  
void operator*(X, float) { }  
  
int main() {  
    X x;  
    int y = 10;  
    float z = 10;  
  
    x * y;  
    x * z;  
}
```

The call `x * y` is interpreted as `x.operator*(y)`. The call `x * z` is interpreted as `operator*(x, z)`.

RELATED REFERENCES

- “Binary Expressions” on page 124

Overloading Assignments

C++ You overload the assignment operator, **`operator=`**, with a nonstatic member function that has only one parameter. You cannot declare an overloaded assignment operator that is a nonmember function. The following example shows how you can overload the assignment operator for a particular class:

```
struct X {  
    int data;  
    X& operator=(X& a) { return a; }  
    X& operator=(int a) {  
        data = a;  
        return *this;  
    }  
};  
  
int main() {  
    X x1, x2;  
    x1 = x2;        // call x1.operator=(x2)  
    x1 = 5;         // call x1.operator=(5)  
}
```

The assignment `x1 = x2` calls `X& X::operator=(X&)`. The assignment `x1 = 5` calls `X& X::operator=(int)`.

Note that because the compiler implicitly declares a copy assignment operator for a class if you do not define one yourself, the copy assignment operator of a derived class will hide the copy assignment operator of its base class. However, you can declare any copy assignment operator as **virtual**. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
    A& operator=(char) {
        cout << "A& A::operator=(char)" << endl;
        return *this;
    }
    virtual A& operator=(const A&) {
        cout << "A& A::operator=(const A&)" << endl;
        return *this;
    }
};

struct B : A {
    B& operator=(char) {
        cout << "B& B::operator=(char)" << endl;
        return *this;
    }
    virtual B& operator=(const A&) {
        cout << "B& B::operator=(const A&)" << endl;
        return *this;
    }
};

struct C : B { };

int main() {
    B b1;
    B b2;
    A* ap1 = &b1;
    A* ap2 = &b1;
    *ap1 = 'z';
    *ap2 = b2;

    C c1;
    // c1 = 'z';
}
```

The following is the output of the above example:

```
A& A::operator=(char)
B& B::operator=(const A&)
```

The assignment `*ap1 = 'z'` calls `A& A::operator=(char)`. Because this operator has not been declared **virtual**, the compiler chooses the function based on the type of the pointer `ap1`. The assignment `*ap2 = b2` calls `B& B::operator=(const A&)`. Because this operator has been declared **virtual**, the compiler chooses the function based on the type of the object that the pointer `ap1` points to. The compiler would not allow the assignment `c1 = 'z'` because the implicitly declared copy assignment operator declared in class `C` hides `B& B::operator=(char)`.

RELATED REFERENCES

- “Member Functions” on page 295
- “Copy Assignment Operators” on page 363

- “Assignment Expressions” on page 134

Overloading Function Calls

C++ The function call operator, when overloaded, does not modify how functions are called. Rather, it modifies how the operator is to be interpreted when applied to objects of a given type.

You overload the function call operator, **operator()**, with a nonstatic member function that has any number of parameters. If you overload a function call operator for a class its declaration will have the following form:

```
return_type operator()(parameter_list)
```

Unlike all other overloaded operators, you can provide default arguments and ellipses in the argument list for the function call operator.

The following example demonstrates how the compiler interprets function call operators:

```
struct A {
    void operator()(int a, char b, ...) { }
    void operator()(char c, int d = 20) { }
};

int main() {
    A a;
    a(5, 'z', 'a', 0);
    a('z');
    // a();
}
```

The function call `a(5, 'z', 'a', 0)` is interpreted as `a.operator()(5, 'z', 'a', 0)`. This calls `void A::operator()(int a, char b, ...)`. The function call `a('z')` is interpreted as `a.operator>('z')`. This calls `void A::operator()(char c, int d = 20)`. The compiler would not allow the function call `a()` because its argument list does not match any function call parameter list defined in class A.

The following example demonstrates an overloaded function call operator:

```
class Point {
private:
    int x, y;
public:
    Point() : x(0), y(0) { }
    Point& operator()(int dx, int dy) {
        x += dx;
        y += dy;
        return *this;
    }
};

int main() {
    Point pt;

    // Offset this coordinate x with 3 points
    // and coordinate y with 2 points.
    pt(3, 2);
}
```

The above example reinterprets the function call operator for objects of class `Point`. If you treat an object of `Point` like a function and pass it two integer arguments, the function call operator will add the values of the arguments you passed to `Point::x` and `Point::y` respectively.

RELATED REFERENCES

- “Function Calls ()” on page 104

Overloading Subscripting

C++ You overload **`operator[]`** with a nonstatic member function that has only one parameter. The following example is a simple array class that has an overloaded subscripting operator. The overloaded subscripting operator throws an exception if you try to access the array outside of its specified bounds:

```
#include <iostream>
using namespace std;

template <class T> class MyArray {
private:
    T* storage;
    int size;
public:
    MyArray(int arg = 10) {
        storage = new T[arg];
        size = arg;
    }

    ~MyArray() {
        delete[] storage;
        storage = 0;
    }

    T& operator[](const int location) throw (const char *);
};

template <class T> T& MyArray<T>::operator[](const int location)
    throw (const char *) {
    if (location < 0 || location >= size) throw "Invalid array access";
    else return storage[location];
}

int main() {
    try {
        MyArray<int> x(13);
        x[0] = 45;
        x[1] = 2435;
        cout << x[0] << endl;
        cout << x[1] << endl;
        x[13] = 84;
    }
    catch (const char* e) {
        cout << e << endl;
    }
}
```

The following is the output of the above example:

```
45
2435
Invalid array access
```

The expression `x[1]` is interpreted as `x.operator[] (1)` and calls `int& MyArray<int>::operator[] (const int)`.

RELATED REFERENCES

- “Array Subscript [] (Array Element Specification)” on page 106

Overloading Class Member Access

C++ You overload **operator->** with a nonstatic member function that has no parameters. The following example demonstrates how the compiler interprets overloaded class member access operators:

```
struct Y {
    void f() { };
};

struct X {
    Y* ptr;
    Y* operator->() {
        return ptr;
    };
};

int main() {
    X x;
    x->f();
}
```

The statement `x->f()` is interpreted as `(x.operator->())->f()`.

The **operator->** is used (often in conjunction with the pointer-dereference operator) to implement “smart pointers.” These pointers are objects that behave like normal pointers except they perform other tasks when you access an object through them, such as automatic object deletion (either when the pointer is destroyed, or the pointer is used to point to another object), or reference counting (counting the number of smart pointers that point to the same object, then automatically deleting the object when that count reaches zero).

One example of a smart pointer is included in the C++ Standard Library called `auto_ptr`. You can find it in the `<memory>` header. The `auto_ptr` class implements automatic object deletion.

RELATED REFERENCES

- “Arrow Operator `->`” on page 107

Overloading Increment and Decrement

C++ You overload the prefix increment operator `++` with either a nonmember function operator that has one argument of class type or a reference to class type, or with a member function operator that has no arguments.

In the following example, the increment operator is overloaded in both ways:

```
class X {
public:

    // member prefix ++x
    void operator++() { }
};

class Y { };

// non-member prefix ++y
void operator++(Y&) { }
```



```

int main() {
    X x;
    Y y;

    // calls x.operator++()
    ++x;

    // explicit call, like ++x
    x.operator++();

    // calls operator++(y)
    ++y;

    // explicit call, like ++y
    operator++(y);
}

```

The postfix increment operator `++` can be overloaded for a class type by declaring a nonmember function `operator operator++()` with two arguments, the first having class type and the second having type **int**. Alternatively, you can declare a member function `operator operator++()` with one argument having type **int**. The compiler uses the **int** argument to distinguish between the prefix and postfix increment operators. For implicit calls, the default value is zero.

For example:

```

class X {
public:

    // member postfix x++
    void operator++(int) { };
};

class Y { };

// nonmember postfix y++
void operator++(Y&, int) { };

int main() {
    X x;
    Y y;

    // calls x.operator++(0)
    // default argument of zero is supplied by compiler
    x++;
    // explicit call to member postfix x++
    x.operator++(0);

    // calls operator++(y, 0)
    y++;

    // explicit call to non-member postfix y++
    operator++(y, 0);
}

```

The prefix and postfix decrement operators follow the same rules as their increment counterparts.

RELATED REFERENCES

- “Member Functions” on page 295
- “Increment `++`” on page 114
- “Decrement `--`” on page 114

Overload Resolution

► C++ The process of selecting the most appropriate overloaded function or operator is called *overload resolution*.

Suppose that `f` is an overloaded function name. When you call the overloaded function `f()`, the compiler creates a set of *candidate functions*. This set of functions includes all of the functions named `f` that can be accessed from the point where you called `f()`. The compiler may include as a candidate function an alternative representation of one of those accessible functions named `f` to facilitate overload resolution.

After creating a set of candidate functions, the compiler creates a set of *viable functions*. This set of functions is a subset of the candidate functions. The number of parameters of each viable function agrees with the number of arguments you used to call `f()`.

The compiler chooses the *best viable function*, the function declaration that the C++ run time will use when you call `f()`, from the set of viable functions. The compiler does this by *implicit conversion sequences*. An implicit conversion sequence is the sequence of conversions required to convert an argument in a function call to the type of the corresponding parameter in a function declaration. The implicit conversion sequences are ranked; some implicit conversion sequences are better than others. The compiler tries to find one viable function in which all of its parameters have either better or equal-ranked implicit conversion sequences than all of the other viable functions. The viable function that the compiler finds is the best viable function. The compiler will not allow a program in which the compiler was able to find more than one best viable function.

You can override an exact match by using an explicit cast. In the following example, the second call to `f()` matches with `f(void*)`:

```
void f(int) { };
void f(void*) { };

int main() {
    f(0xaabb);           // matches f(int);
    f((void*) 0xaabb);    // matches f(void*)
}
```

RELATED REFERENCES

- “Implicit Conversion Sequences”

Implicit Conversion Sequences

► C++ An *implicit conversion sequence* is the sequence of conversions required to convert an argument in a function call to the type of the corresponding parameter in a function declaration.

The compiler will try to determine an implicit conversion sequence for each argument. It will then categorize each implicit conversion sequence in one of three categories and rank them depending on the category. The compiler will not allow any program in which it cannot find an implicit conversion sequence for an argument.

The following are the three categories of conversion sequences in order from best to worst:

- Standard conversion sequences
- User-defined conversion sequences
- Ellipsis conversion sequences

Note: Two standard conversion sequences or two user-defined conversion sequences may have different ranks.

Standard Conversion Sequences

Standard conversion sequences are categorized in one of three ranks. The ranks are listed in order from best to worst:

- Exact match: This rank includes the following conversions:
 - Identity conversions
 - Lvalue-to-rvalue conversions
 - Array-to-pointer conversions
 - Qualification conversions
- Promotion: This rank includes integral and floating point promotions.
- Conversion: This rank includes the following conversions:
 - Integral and floating-point conversions
 - Floating-integral conversions
 - Pointer conversions
 - Pointer-to-member conversions
 - Boolean conversions

The compiler ranks a standard conversion sequence by its worst-ranked standard conversion. For example, if a standard conversion sequence has a floating-point conversion, then that sequence has conversion rank.

User-Defined Conversion Sequences

A *user-defined conversion sequence* consists of the following:

- A standard conversion sequence
- A user-defined conversion
- A second standard conversion sequence

A user-defined conversion sequence A is better than a user-defined conversion sequence B if the both have the same user-defined conversion function or constructor, and the second standard conversion sequence of A is better than the second standard conversion sequence of B.

Ellipsis Conversion Sequences

An *ellipsis conversion sequence* occurs when the compiler matches an argument in a function call with a corresponding ellipsis parameter.

RELATED REFERENCES

- “Lvalue-to-Rvalue Conversions” on page 145
- “Pointer Conversions” on page 146
- “Qualification Conversions” on page 148
- “Integral Conversions” on page 145
- “Floating-Point Conversions” on page 146
- “Boolean Conversions” on page 145

Resolving Addresses of Overloaded Functions

C++ If you use an overloaded function name `f` without any arguments, that name can refer to a function, a pointer to a function, a pointer to member function, or a specialization of a function template. Because you did not provide any arguments, the compiler cannot perform overload resolution the same way it would for a function call or for the use of an operator. Instead, the compiler will try to choose the best viable function that matches the type of one of the following expressions, depending on where you have used `f`:

- An object or reference you are initializing
- The left side of an assignment
- A parameter of a function or a user-defined operator
- The return value of a function, operator, or conversion
- An explicit type conversion

If the compiler chose a declaration of a nonmember function or a static member function when you used `f`, the compiler matched the declaration with an expression of type pointer-to-function or reference-to-function. If the compiler chose a declaration of a nonstatic member function, the compiler matched that declaration with an expression of type pointer-to-member function. The following example demonstrates this:

```
struct X {
    int f(int) { return 0; }
    static int f(char) { return 0; };
}

int main() {
    int (X::*a)(int) = &X::f;
    // int (*b)(int) = &X::f;
    int (*c)(int) = &X::f;
}
```

The compiler will not allow the initialization of the function pointer `b`. No nonmember function or static function of type **`int(int)`** has been declared.

If `f` is a template function, the compiler will perform template argument deduction to determine which template function to use. If successful, it will add that function to the list of viable functions. If there is more than one function in this set, including a non-template function, the compiler will eliminate all template functions from the set. If there are only template functions in this set, the compiler will choose the most specialized template function. The following example demonstrates this:

```
template<class T> int f(T) { return 0; }
template<> int f(int) { return 0; }
int f(int) { return 0; }

int main() {
    int (*a)(int) = f;
    a(1);
}
```

The function call `a(1)` calls `int f(int)`.

RELATED REFERENCES

- “Chapter 7. Functions” on page 153
- “Pointers to Functions” on page 173
- “Pointers to Members” on page 298
- “Function Templates” on page 379
- “Explicit Specialization” on page 390

Chapter 12. Classes

C++ A *class* is a mechanism for creating user-defined data types. It is similar to the C-language structure data type. In C, a structure is composed of a set of data members. In C++, a class type is like a C structure, except that a class is composed of a set of data members and a set of operations that can be performed on the class.

In C++, a class type can be declared with the keywords **union**, **struct**, or **class**. A union object can hold any one of a set of named members. Structure and class objects hold a complete set of members. Each class type represents a unique set of class members including data members, member functions, and other type names. The default access for members depends on the class key:

- The members of a class declared with the keyword **class** are private by default. A class is inherited privately by default.
- The members of a class declared with the keyword **struct** are public by default. A structure is inherited publicly by default.
- The members of a union (declared with the keyword **union**) are public by default. A union cannot be used as a base class in derivation.

Once you create a class type, you can declare one or more objects of that class type. For example:

```
class X
{
    /* define class members here */
};
int main()
{
    X xobject1;      // create an object of class type X
    X xobject2;      // create another object of class type X
}
```

You may have *polymorphic* classes in C++. Polymorphism is the ability to use a function name that appears in different classes (related by inheritance), without knowing exactly the class the function belongs to at compile time.

C++ allows you to redefine standard operators and functions through the concept of overloading. Operator overloading facilitates data abstraction by allowing you to use classes as easily as built-in types.

RELATED REFERENCES

- “Structures” on page 51
- “Chapter 13. Class Members and Friends” on page 293
- “Chapter 14. Inheritance” on page 315
- “Chapter 11. Overloading” on page 269
- “Virtual Functions” on page 333

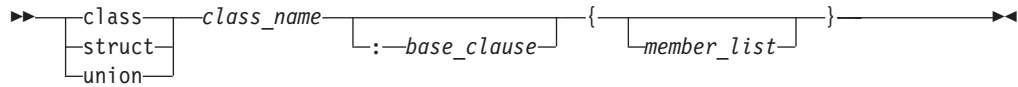
Declaring Class Types

C++ A class declaration creates a unique type class name.

Declaring Class Objects

A *class specifier* is a type specifier used to declare a class. Once a class specifier has been seen and its members declared, a class is considered to be defined even if the member functions of that class are not yet defined. A class specifier has the following form:

Syntax — Class Specifier



The *class_name* is a unique identifier that becomes a reserved word within its scope. Once a class name is declared, it hides other declarations of the same name within the enclosing scope.

The *member_list* specifies the class members, both data and functions, of the class *class_name*. If the *member_list* of a class is empty, objects of that class have a nonzero size. You can use a *class_name* within the *member_list* of the class specifier itself as long as the size of the class is not required.

The *base_clause* specifies the base class or classes from which the class *class_name* inherits members. If the *base_clause* is not empty, the class *class_name* is called a *derived class*.

A *structure* is a class declared with the *class_key* **struct**. The members and base classes of a structure are public by default. A *union* is a class declared with the *class_key* **union**. The members of a union are public by default; a union holds only one data member at a time.

An *aggregate class* is a class that has no user-defined constructors, no private or protected non-static data members, no base classes, and no virtual functions.

RELATED REFERENCES

- “Class Member Lists” on page 293
- “Derivation” on page 317

Using Class Objects

C++ You can use a class type to create instances or *objects* of that class type. For example, you can declare a class, structure, and union with class names X, Y, and Z respectively:

```
class X {
    // members of class X
};

struct Y {
    // members of struct Y
};

union Z {
    // members of union Z
};
```

You can then declare objects of each of these class types. Remember that classes, structures, and unions are all types of C++ classes.

```
int main()
{
    X xobj;      // declare a class object of class type X
    Y yobj;      // declare a struct object of class type Y
    Z zobj;      // declare a union object of class type Z
}
```

In C++, unlike C, you do not need to precede declarations of class objects with the keywords **union**, **struct**, and **class** unless the name of the class is hidden. For example:

```
struct Y { /* ... */ };
class X { /* ... */ };
int main ()
{
    int X;          // hides the class name X
    Y yobj;         // valid
    X xobj;         // error, class name X is hidden
    class X xobj;   // valid
}
```

When you declare more than one class object in a declaration, the declarators are treated as if declared individually. For example, if you declare two objects of class S in a single declaration:

```
class S { /* ... */ };
int main()
{
    S S,T; // declare two objects of class type S
}
```

this declaration is equivalent to:

```
class S { /* ... */ };
int main()
{
    S S;
    class S T;      // keyword class is required
                   // since variable S hides class type S
}
```

but is not equivalent to:

```
class S { /* ... */ };
int main()
{
    S S;
    S T;            // error, S class type is hidden
}
```

You can also declare references to classes, pointers to classes, and arrays of classes. For example:

```
class X { /* ... */ };
struct Y { /* ... */ };
union Z { /* ... */ };
int main()
{
    X xobj;
    X &xref = xobj;    // reference to class object of type X
    Y *yptr;          // pointer to struct object of type Y
    Z zarray[10];     // array of 10 union objects of type Z
}
```

Objects of class types that are not copy restricted can be assigned, passed as arguments to functions, and returned by functions.

Declaring Class Objects

RELATED REFERENCES

- “Chapter 12. Classes” on page 283
- “Structures” on page 51
- “Unions” on page 59
- “References” on page 92
- “Scope of Class Names” on page 287

Classes and Structures

C++ The C++ class is an extension of the C-language structure. Because the only difference between a structure and a class is that structure members have public access by default and a class members have private access by default, you can use the keywords **class** or **struct** to define equivalent classes.

For example, in the following code fragment, the class X is equivalent to the structure Y:

CCNX10C

```
class X {  
  
    // private by default  
    int a;  
  
public:  
  
    // public member function  
    int f() { return a = 5; };  
};  
  
struct Y {  
  
    // public by default  
    int f() { return a = 5; };  
  
private:  
  
    // private data member  
    int a;  
};
```

If you define a structure and then declare an object of that structure using the keyword **class**, the members of the object are still public by default. In the following example, `main()` has access to the members of `obj_X` even though `obj_X` has been declared using an elaborated type specifier that uses the class key **class**:

CCNX10D

```
#include <iostream>  
using namespace std;  
  
struct X {  
    int a;  
    int b;  
};  
  
class X obj_X;  
  
int main() {
```



```
obj_X.a = 0;
obj_X.b = 1;
cout << "Here are a and b: " << obj_X.a << " " << obj_X.b << endl;
}
```

The following is the output of the above example:

Here are a and b: 0 1

RELATED REFERENCES

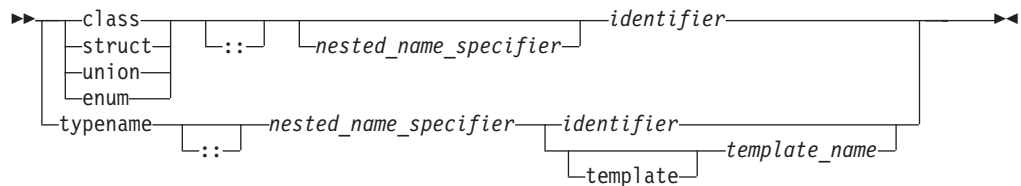
- “Structures” on page 51

Scope of Class Names

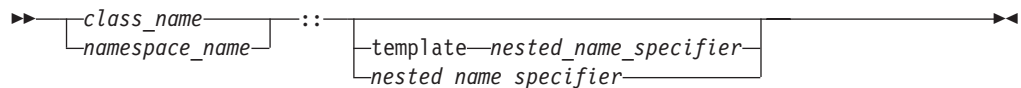
C++ A class declaration introduces the class name into the scope where it is declared. Any class, object, function or other declaration of that name in an enclosing scope is hidden.

If a class name is declared in the same scope as a function, enumerator, or object with the same name, you must refer to that class using an *elaborated type specifier*.

Syntax — Elaborated Type Specifier



Syntax — Nested Name Specifier



The following example must use an elaborated type specifier to refer to class A because this class is hidden by the definition of the function A():

```
class A { };

void A (class A*) { };

int main()
{
    class A* x;
    A(x);
}
```

The declaration `class A* x` is an elaborated type specifier. Declaring a class with the same name of another function, enumerator, or object as demonstrated above is not recommended.

An elaborated type specifier can also be used in the incomplete declaration of a class type to reserve the name for a class type within the current scope.

Scope of Class Names

RELATED REFERENCES

- “Scope” on page 1
- “Incomplete Class Declarations”

Incomplete Class Declarations

C++ An *incomplete class declaration* is a class declaration that does not define any class members. You cannot declare any objects of the class type or refer to the members of a class until the declaration is complete. However, an incomplete declaration allows you to make specific references to a class prior to its definition as long as the size of the class is not required.

For example, you can define a pointer to the structure first in the definition of the structure second. Structure first is declared in an incomplete class declaration prior to the definition of second, and the definition of `oneptr` in structure second does not require the size of first:

```
struct first;           // incomplete declaration of struct first

struct second           // complete declaration of struct second
{
    first* oneptr;       // pointer to struct first refers to
                        // struct first prior to its complete
                        // declaration

    first one;           // error, you cannot declare an object of
                        // an incompletely declared class type

    int x, y;
};

struct first            // complete declaration of struct first
{
    second two;         // define an object of class type second
    int z;
};
```

However, if you declare a class with an empty member list, it is a complete class declaration. For example:

```
class X;                // incomplete class declaration
class Z {};             // empty member list
class Y
{
public:
    X yobj;             // error, cannot create an object of an
                        // incomplete class type

    Z zobj;             // valid
};
```

RELATED REFERENCES

- “Class Member Lists” on page 293

Nested Classes

C++ A *nested class* is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class. The following example demonstrates this:

```
class A {
    int x;

    class B { };

    class C {

        // The compiler cannot allow the following
        // declaration because A::B is private:
        //   B b;

        int y;
        void f(A* p, int i) {

            // The compiler cannot allow the following
            // statement because A::x is private:
            //   p->x = i;

        }
    };

    void g(C* p) {

        // The compiler cannot allow the following
        // statement because C::y is private:
        //   int z = p->y;
    }
};

int main() { }
```

The compiler would not allow the declaration of object `b` because class `A::B` is private. The compiler would not allow the statement `p->x = i` because `A::x` is private. The compiler would not allow the statement `int z = p->y` because `C::y` is private.

You can define member functions and static data members of a nested class in namespace scope. For example, in the following code fragment, you can access the static members `x` and `y` and member functions `f()` and `g()` of the nested class nested by using a qualified type name. Qualified type names allow you to define a **typedef** to represent a qualified class name. You can then use the **typedef** with the `::` (scope resolution) operator to refer to a nested class or class member, as shown in the following example:

```
class outside
{
public:
    class nested
    {
    public:
        static int x;
        static int y;
        int f();
        int g();
    };
};

int outside::nested::x = 5;
int outside::nested::f() { return 0; };
```

Scope of Class Names

```
typedef outside::nested outnest;    // define a typedef
int outnest::y = 10;                // use typedef with ::
int outnest::g() { return 0; };
```

However, using a typedef to represent a nested class name hides information and may make the code harder to understand.

You cannot use a typedef name in an elaborated type specifier. To illustrate, you cannot use the following declaration in the above example:

```
class outnest obj;
```

A nested class may inherit from private members of its enclosing class. The following example demonstrates this:

```
class A {
private:
    class B { };
    B *z;

    class C : private B {
private:
        B y;
    //      A::B y2;
        C *x;
    //      A::C *x2;
    };
};
```

The nested class `A::C` inherits from `A::B`. The compiler does not allow the declarations `A::B y2` and `A::C *x2` because both `A::B` and `A::C` are private.

RELATED REFERENCES

- “Scope of Class Names” on page 287
- “Member Functions” on page 295
- “Member Access” on page 308
- “Static Members” on page 303
- “typedef” on page 43
- “C++ Scope Resolution Operator `::`” on page 102

Local Classes

C++ A *local class* is declared within a function definition. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope, as well as external variables and functions.

For example:

```
int x;                                // global variable
void f()                             // function definition
{
    static int y;                    // static variable y can be used by
                                    // local class
    int x;                          // auto variable x cannot be used by
                                    // local class
    extern int g();                  // extern function g can be used by
                                    // local class

    class local                      // local class
    {
        int g() { return x; }        // error, local variable x
                                    // cannot be used by g
    }
```

```

        int h() { return y; }      // valid, static variable y
        int k() { return ::x; }    // valid, global x
        int l() { return g(); }    // valid, extern function g
    };
}

int main()
{
    local* z;                      // error: the class local is not visible
    // ...}

```

Member functions of a local class have to be defined within their class definition. As a result, member functions of a local class are inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword **inline**.

A local class cannot have static data members. In the following example, an attempt to define a static member of a local class causes an error:

```

void f()
{
    class local
    {
        int f();                // error, local class has nonlinear
                                // member function
        int g() {return 0;}      // valid, inline member function
        static int a;           // error, static is not allowed for
                                // local class
        int b;                  // valid, nonstatic variable
    };
}
// . . .

```

An enclosing function has no special access to members of the local class.

RELATED REFERENCES

- “Member Functions” on page 295
- “Inline Functions” on page 174

Local Type Names

C++ Local type names follow the same scope rules as other names. Type names defined within a class declaration have class scope and cannot be used outside their class without qualification.

If you use a class name, **typedef** name, or a constant name that is used in a type name, in a class declaration, you cannot redefine that name after it is used in the class declaration.

For example:

```

int main ()
{
    typedef double db;
    struct st
    {
        db x;
        typedef int db; // error
        db y;
    };
}

```

Scope of Class Names

The following declarations are valid:

```
typedef float T;
class s {
    typedef int T;
    void f(const T);
};
```

Here, function `f()` takes an argument of type `s::T`. However, the following declarations, where the order of the members of `s` has been reversed, cause an error:

```
typedef float T;
class s {
    void f(const T);
    typedef int T;
};
```

In a class declaration, you cannot redefine a name that is not a class name, or a **typedef** name to a class name or **typedef** name once you have used that name in the class declaration.

RELATED REFERENCES

- “Scope” on page 1
- “Global Scope” on page 2
- “typedef” on page 43

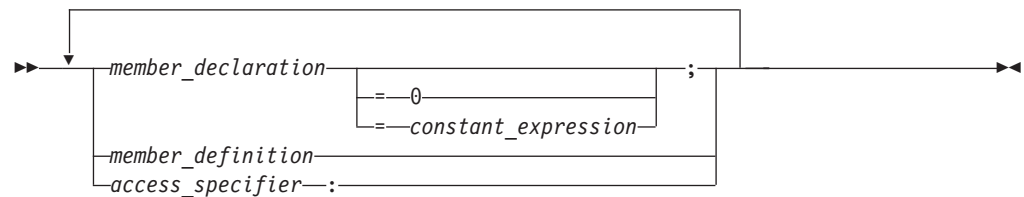
Chapter 13. Class Members and Friends

The concept of information hiding comprises a public class interface and private implementation. It is the mechanism for limiting direct access to the internal representation of a class type by functions in a program. This section discusses the declaration of class members with respect to the information hiding mechanism and how a class can grant functions access to its nonpublic members by the use of the friend mechanism.

Class Member Lists

C++ An optional *member list* declares sub-objects called *class members*. Class members can be data, functions, nested types, and enumerators.

Syntax — Class Member List



The member list follows the class name and is placed between braces. The following applies to member lists, and members of member lists:

- A *member_declaration* or a *member_definition* may be a declaration or definition of a data member, member function, nested type, or enumeration. (The enumerators of an enumeration defined in a class member list are also members of the class.)
- A member list is the only place where you can declare class members.
- Friend declarations are not class members but must appear in member lists.
- The member list in a class definition declares all the members of a class; you cannot add members elsewhere.
- You cannot declare a member twice in a member list.
- You may declare a data member or member function as **static** but not **auto**, **extern**, or **register**.
- You may declare a nested class, a member class template, or a member function, and then define it later outside the class.
- You must define static data members later outside the class.
- Nonstatic members that are class objects must be objects of previously defined classes; a class A cannot contain an object of class A, but it can contain a pointer or reference to an object of class A.
- You must specify all dimensions of a nonstatic array member.

A *constant initializer* (= *constant_expression*) may only appear in a class member of integral or enumeration type that has been declared as **static**.

A *pure specifier* (= 0) indicates that a function has no definition. It is only used with member functions declared as **virtual** and replaces the function definition of a member function in the member list.

An *access specifier* is one of **public**, **private**, or **protected**.

Class Member Lists

A *member declaration* declares a class member for the class containing the declaration.

>VAC++ The order of allocation of nonstatic class members separated by an *access_specifier* is implementation dependent. The VisualAge C++ compiler allocates class members in the order that they are declared.

Suppose A is a name of a class. The following class members of A must have a name different from A:

- All data members
- All type members
- All enumerators of enumerated type members
- All members of all anonymous union members

RELATED REFERENCES

- “Chapter 3. Declarations” on page 33
- “Declaring Class Types” on page 283
- “Member Access” on page 308
- “Virtual Functions” on page 333
- “Static Members” on page 303

Data Members

> C++ Data members include members that are declared with any of the fundamental types, as well as other types, including pointer, reference, array types, bit fields, and user-defined types. You can declare a data member the same way as a variable, except that explicit initializers are not allowed inside the class definition. However, a const static data member of integral or enumeration type may have an explicit initializer.

If an array is declared as a nonstatic class member, you must specify all of the dimensions of the array.

A class can have members that are of a class type or are pointers or references to a class type. Members that are of a class type must be of a class type that is previously declared. An incomplete class type can be used in a member declaration as long as the size of the class is not needed. For example, a member can be declared that is a pointer to an incomplete class type.

A class X cannot have a member that is of type X, but it can contain pointers to X, references to X, and static objects of X. Member functions of X can take arguments of type X and have a return type of X. For example:

```
class X
{
    X();
    X *xptr;
    X &xref;
    static X xcount;
    X xfunc(X);
};
```

RELATED REFERENCES

- “Pointers” on page 81
- “References” on page 92
- “Arrays” on page 86

- “Pointers” on page 81
- “References” on page 92
- “Incomplete Class Declarations” on page 288

Member Functions

► **C++** *Member functions* are operators and functions that are declared as members of a class. Member functions do not include operators and functions declared with the **friend** specifier. These are called *friends* of a class. You can declare a member function as **static**; this is called a *static member function*. A member function that is not declared as **static** is called a *nonstatic member function*.

Suppose that you create an object named *x* of class *A*, and class *A* has a nonstatic member function *f()*. If you call the function *x.f()*, the keyword **this** in the body of *f()* is the address of *x*.

The definition of a member function is within the scope of its enclosing class. The body of a member function is analyzed after the class declaration so that members of that class can be used in the member function body, even if the member function definition appears before the declaration of that member in the class member list. When the function *add()* is called in the following example, the data variables *a*, *b*, and *c* can be used in the body of *add()*.

```
class x
{
public:
    int add()           // inline member function add
    {return a+b+c;};
private:
    int a,b,c;
};
```

Inline Member Functions

You may either define a member function inside its class definition, or you may it outside if you have already declared (but not defined) the member function in the class definition.

A member function that is defined inside its class member list is called an *inline member function*. Member functions containing a few lines of code are usually declared inline. In the above example, *add()* is an inline member function. If you define a member function outside of its class definition, it must appear in a namespace scope enclosing the class definition. You must also qualify the member function name using the scope resolution (*::*) operator.

An equivalent way to declare an inline member function is to either declare it in the class with the **inline** keyword (and define the function outside of its class) or to define it outside of the class declaration using the **inline** keyword.

In the following example, member function *Y::f()* is an inline member function:

```
struct Y {
private:
    char a*;
public:
    char* f() { return a; }
};
```

Member Functions

The following example is equivalent to the previous example; `Y::f()` is an inline member function:

```
struct Y {  
    private:  
        char a*;  
    public:  
        char* f();  
};  
  
inline char* Y::f() { return a; }
```

Inline member functions have internal linkage. Noninline member functions have external linkage.

Member Functions of Local Classes

Member functions of a local class have to be defined within their class definition. As a result, member functions of a local class are inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword **inline**.

RELATED REFERENCES

- “Friends” on page 310
- “Static Member Functions” on page 306
- “Chapter 7. Functions” on page 153
- “Inline Functions” on page 174
- “Local Classes” on page 290

const and volatile Member Functions

C++ A member function declared with the **const** qualifier can be called for constant and nonconstant objects. A nonconstant member function can only be called for a nonconstant object. Similarly, a member function declared with the **volatile** qualifier can be called for volatile and nonvolatile objects. A nonvolatile member function can only be called for a nonvolatile object.

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69

Virtual Member Functions

C++ Virtual member functions are declared with the keyword **virtual**. They allow dynamic binding of member functions. Because all virtual functions must be member functions, virtual member functions are simply called *virtual functions*.

If the definition of a virtual function is replaced by a pure specifier in the declaration of the function, the function is said to be declared pure. A class that has at least one pure virtual function is called an *abstract class*.

RELATED REFERENCES

- “Virtual Functions” on page 333
- “Abstract Classes” on page 339

Special Member Functions

C++ *Special member functions* are used to create, destroy, initialize, convert, and copy class objects. These include the following:

- Constructors
- Destructors
- Conversion constructors
- Conversion functions
- Copy constructors

RELATED REFERENCES

- “Constructors” on page 342
- “Destructors” on page 350
- “Conversion by Constructor” on page 360
- “Conversion Functions” on page 361
- “Copy Constructors” on page 362

Member Scope

C++ Member functions and static members can be defined outside their class declaration if they have already been declared, but not defined, in the class member list. Nonstatic data members are defined when an object of their class is created. The declaration of a static data member is not a definition. The declaration of a member function is a definition if the body of the function is also given.

Whenever the definition of a class member appears outside of the class declaration, the member name must be qualified by the class name using the `::` (scope resolution) operator.

The following example defines a member function outside of its class declaration.

CCNX11A

```
#include <iostream>
using namespace std;

struct X {
    int a, b ;

    // member function declaration only
    int add();
};

// global variable
int a = 10;

// define member function outside its class declaration
int X::add() { return a + b; }

int main() {
    int answer;
    X xobject;
    xobject.a = 1;
    xobject.b = 2;
    answer = xobject.add();
    cout << xobject.a << " + " << xobject.b << " = " << answer << endl;
}
```

The output for this example is: 1 + 2 = 3

Member Scope

All member functions are in class scope even if they are defined outside their class declaration. In the above example, the member function `add()` returns the data member `a`, not the global variable `a`.

The name of a class member is local to its class. Unless you use one of the class access operators, `.` (dot), or `->` (arrow), or `::` (scope resolution) operator, you can only use a class member in a member function of its class and in nested classes. You can only use types, enumerations and static members in a nested class without qualification with the `::` operator.

The order of search for a name in a member function body is:

1. Within the member function body itself
2. Within all the enclosing classes, including inherited members of those classes
3. Within the lexical scope of the body declaration

The search of the enclosing classes, including inherited members, is demonstrated in the following example:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class Z : A {
    class Y : B {
        class X : C { int f(); /* ... */ };
    };
};
int Z::Y::X f()
{
    char j;
    return 0;
}
```

In this example, the search for the name `j` in the definition of the function `f` follows this order:

1. In the body of the function `f`
2. In `X` and in its base class `C`
3. In `Y` and in its base class `B`
4. In `Z` and in its base class `A`
5. In the lexical scope of the body of `f`. In this case, this is global scope.

Note that when the containing classes are being searched, only the definitions of the containing classes and their base classes are searched. The scope containing the base class definitions (global scope, in this example) is not searched.

RELATED REFERENCES

- “Class Member Lists” on page 293
- “C++ Scope Resolution Operator `::`” on page 102
- “Class Scope” on page 3

Pointers to Members

C++ Pointers to members allow you to refer to nonstatic members of class objects. You cannot use a pointer to member to point to a static class member because the address of a static member is not associated with any particular object. To point to a static class member, you must use a normal pointer.

You can use pointers to member functions in the same manner as pointers to functions. You can compare pointers to member functions, assign values to them, and use them to call member functions. Note that a member function does not have the same type as a nonmember function that has the same number and type of arguments and the same return type.

Pointers to members can be declared and used as shown in the following example:

```
#include <iostream>
using namespace std;

class X {
public:
    int a;
    void f(int b) {
        cout << "The value of b is " << b << endl;
    }
};

int main() {

    // declare pointer to data member
    int X::*ptiptr = &X::a;

    // declare a pointer to member function
    void (X::* ptfptr) (int) = &X::f;

    // create an object of class type X
    X xobject;

    // initialize data member
    xobject.*ptiptr = 10;

    cout << "The value of a is " << xobject.*ptiptr << endl;

    // call member function
    (xobject.*ptfptr) (20);
}
```

The output for this example is:

```
The value of a is 10
The value of b is 20
```

To reduce complex syntax, you can declare a **typedef** to be a pointer to a member. A pointer to a member can be declared and used as shown in the following code fragment:

```
typedef int X::*my_pointer_to_member;
typedef void (X::*my_pointer_to_function) (int);

int main() {
    my_pointer_to_member ptiptr = &X::a;
    my_pointer_to_function ptfptr = &X::f;
    X xobject;
    xobject.*ptiptr = 10;
    cout << "The value of a is " << xobject.*ptiptr << endl;
    (xobject.*ptfptr) (20);
}
```

The pointer to member operators `.*` and `->*` are used to bind a pointer to a member of a specific class object. Because the precedence of `()` (function call operator) is higher than `.*` and `->*`, you must use parentheses to call the function pointed to by `ptf`.

Pointers to Members

For more information, see “C++ Pointer to Member Operators . * ->*” on page 133.

RELATED REFERENCES

- “Static Members” on page 303
- “typedef” on page 43
- “C++ Pointer to Member Operators . * ->*” on page 133

The this Pointer

C++ The keyword **this** identifies a special type of pointer. Suppose that you create an object named *x* of class *A*, and class *A* has a nonstatic member function *f()*. If you call the function *x.f()*, the keyword **this** in the body of *f()* is the address of *x*. You cannot declare the **this** pointer or make assignments to it.

A static member function does not have a **this** pointer.

The type of the **this** pointer for a member function of a class type *X*, is *X* const*. If the member function is declared with the **const** qualifier, the type of the **this** pointer for that member function for class *X*, is *const X* const*. If the member function is declared with the **volatile** qualifier, the type of the **this** pointer for that member function for class *X* is *volatile X* const*. For example, the compiler will not allow the following:

```
struct A {  
    int a;  
    int f() const { return a++; }  
};
```

The compiler will not allow the statement *a++* in the body of function *f()*. In the function *f()*, the **this** pointer is of type *A* const*. The function *f()* is trying to modify part of the object to which **this** points.

The **this** pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.

For example, you can refer to the particular class object that a member function is called for by using the **this** pointer in the body of the member function. The following code example produces the output *a = 5*:

CCNX11C

```
#include <iostream>  
using namespace std;  
  
struct X {  
private:  
    int a;  
public:  
    void Set_a(int a) {  
  
        // The 'this' pointer is used to retrieve 'xobj.a'  
        // hidden by the automatic variable 'a'  
        this->a = a;  
    }  
    void Print_a() { cout << "a = " << a << endl; }  
};  
  
int main() {  
    X xobj;
```

```
int a = 5;
xobj.Set_a(a);
xobj.Print_a();
}
```

In the member function `Set_a()`, the statement `this->a = a` uses the **this** pointer to retrieve `xobj.a` hidden by the automatic variable `a`.

Unless a class member name is hidden, using the class member name is equivalent to using the class member name with the **this** pointer and the class member access operator (`->`).

The example in the first column of the following table shows code that uses class members without the **this** pointer. The code in the second column uses the variable `THIS` to simulate the first column's hidden use of the **this** pointer:

Code without using this pointer	Equivalent code, the <code>THIS</code> variable simulating the hidden use of the this pointer
--	--

The this Pointer

<pre>#include <string> #include <iostream> using namespace std; struct X { private: int len; char *ptr; public: int GetLen() { return len; } char * GetPtr() { return ptr; } X& Set(char *); X& Cat(char *); X& Copy(X&); void Print(); }; X& X::Set(char *pc) { len = strlen(pc); ptr = new char[len]; strcpy(ptr, pc); return *this; } X& X::Cat(char *pc) { len += strlen(pc); strcat(ptr, pc); return *this; } X& X::Copy(X& x) { Set(x.GetPtr()); return *this; } void X::Print() { cout << ptr << endl; } int main() { X xobj1; xobj1.Set("abcd") .Cat("efgh"); xobj1.Print(); X xobj2; xobj2.Copy(xobj1) .Cat("ijkl"); xobj2.Print(); }</pre>	<pre>#include <string> #include <iostream> using namespace std; struct X { private: int len; char *ptr; public: int GetLen (X* const THIS) { return THIS->len; } char * GetPtr (X* const THIS) { return THIS->ptr; } X& Set(X* const, char *); X& Cat(X* const, char *); X& Copy(X* const, X&); void Print(X* const); }; X& X::Set(X* const THIS, char *pc) { THIS->len = strlen(pc); THIS->ptr = new char[THIS->len]; strcpy(THIS->ptr, pc); return *THIS; } X& X::Cat(X* const THIS, char *pc) { THIS->len += strlen(pc); strcat(THIS->ptr, pc); return *THIS; } X& X::Copy(X* const THIS, X& x) { THIS->Set(THIS, x.GetPtr(&x)); return *THIS; } void X::Print(X* const THIS) { cout << THIS->ptr << endl; } int main() { X xobj1; xobj1.Set(&xobj1 , "abcd") .Cat(&xobj1 , "efgh"); xobj1.Print(&xobj1); X xobj2; xobj2.Copy(&xobj2 , xobj1) .Cat(&xobj2 , "ijkl"); xobj2.Print(&xobj2); }</pre>
--	--

Both examples produces the following output:

```
abcdefgh
abcdefghijkl
```

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69

Static Members

C++ Class members can be declared using the storage-class specifier **static** in the class member list. Only one copy of the static member is shared by all objects of a class in a program. When you declare an object of a class having a static member, the static member is not part of the class object.

A typical use of static members is for recording data common to all objects of a class. For example, you can use a static data member as a counter to store the number of objects of a particular class type that are created. Each time a new object is created, this static data member can be incremented to keep track of the total number of objects.

You access a static member by qualifying the class name using the `::` (scope resolution) operator. In the following example, you can refer to the static member `f()` of class type `X` as `X::f()` even if no object of type `X` is ever declared:

```
struct X {
    static int f();
};

int main() {
    X::f();
}
```

For more information on the storage-class specifier **static**, see “static Storage Class Specifier” on page 42

RELATED REFERENCES

- “static Storage Class Specifier” on page 42
- “Class Member Lists” on page 293

Using the Class Access Operators with Static Members

C++ You do not have to use the class member access syntax to refer to a static member; to access a static member `s` of class `X`, you could use the expression `X::s`. The following example demonstrates accessing a static member:

```
#include <iostream>
using namespace std;

struct A {
    static void f() { cout << "In static function A::f()" << endl; }
};

int main() {

    // no object required for static member
    A::f();

    A a;
    A* ap = &a;
    a.f();
    ap->f();
}
```

The three statements `A::f()`, `a.f()`, and `ap->f()` all call the same static member function `A::f()`.

Static Members

You can directly refer to a static member in the same scope of its class, or in the scope of a class derived from the static member's class. The following example demonstrates the latter case (directly referring to a static member in the scope of a class derived from the static member's class):

```
#include <iostream>
using namespace std;

int g() {
    cout << "In function g()" << endl;
    return 0;
}

class X {
public:
    static int g() {
        cout << "In static member function X::g()" << endl;
        return 1;
    }
};

class Y: public X {
public:
    static int i;
};

int Y::i = g();

int main() { }
```

The following is the output of the above code:

In static member function X::g()

The initialization `int Y::i = g()` calls `X::g()`, not the function `g()` declared in the global namespace.

A static member can be referred to independently of any association with a class object because there is only one static member shared by all objects of a class. A static member will exist even if no objects of its class have been declared.

RELATED REFERENCES

- “static Storage Class Specifier” on page 42
- “C++ Scope Resolution Operator `::`” on page 102
- “Dot Operator `.`” on page 107
- “Arrow Operator `->`” on page 107

Static Data Members

C++ Only one copy of a static data member of a class exists; it is shared with all objects of that class.

Static data members of a class in namespace scope have external linkage. Static data members follow the usual class access rules, except that they can be initialized in file scope. Static data members and their initializers can access other static private and protected members of their class. The initializer for a static data member is in the scope of the class declaring the member.

A static data member can be of any type except for **void** or **void** qualified with **const** or **volatile**.

The declaration of a static data member in the member list of a class is not a definition. The definition of a static data member is equivalent to an external variable definition. You must define the static member outside of the class declaration in namespace scope.

For example:

```
class X
{
public:
    static int i;
};
int X::i = 0; // definition outside class declaration
```

Once you define a static data member, it exists even though no objects of the static data member's class exist. In the above example, no objects of class `X` exist even though the static data member `X::i` has been defined.

The following example shows how you can initialize static members using other static members, even though these members are private:

```
class C {
    static int i;
    static int j;
    static int k;
    static int l;
    static int m;
    static int n;
    static int p;
    static int q;
    static int r;
    static int s;
    static int f() { return 0; }
    int a;
public:
    C() { a = 0; }
};

C c;
int C::i = C::f(); // initialize with static member function
int C::j = C::i;   // initialize with another static data member
int C::k = c.f();  // initialize with member function from an object
int C::l = c.j;    // initialize with data member from an object
int C::s = c.a;    // initialize with nonstatic data member
int C::r = 1;      // initialize with a constant value

class Y : private C { } y;

int C::m = Y::f();
int C::n = Y::r;
int C::p = y.r;    // error
int C::q = y.f();  // error
```

The initializations of `C::p` and `C::x` cause errors because `y` is an object of a class that is derived privately from `C`, and its members are not accessible to members of `C`.

If a static data member is of **const** integral or **const** enumeration type, you may specify a *constant initializer* in the static data member's declaration. This constant initializer must be an integral constant expression. Note that the constant initializer is not a definition. You still need to define the static member in an enclosing namespace is still required. The following example demonstrates this:

```
#include <iostream>
using namespace std;
```

Static Members

```
struct X {
    static const int a = 76;
};

const int X::a;

int main() {
    cout << X::a << endl;
}
```

The tokens = 76 at the end of the declaration of static data member a is a constant initializer.

You can only have one definition of a static member in a program. Unnamed classes and classes contained within unnamed classes cannot have static data members.

You cannot declare a static data member as **mutable**.

Local classes cannot have static data members.

RELATED REFERENCES

- “External Linkage” on page 6
- “Member Access” on page 308
- “Local Classes” on page 290

Static Member Functions

C++ You cannot have static and nonstatic member functions with the same names and the same number and type of arguments.

Like static data members, you may access a static member function `f()` of a class A without using an object of class A.

A static member function does not have a **this** pointer. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct X {
private:
    int i;
    static int si;
public:
    void set_i(int arg) { i = arg; }
    static void set_si(int arg) { si = arg; }

    void print_i() {
        cout << "Value of i = " << i << endl;
        cout << "Again, value of i = " << this->i << endl;
    }

    static void print_si() {
        cout << "Value of si = " << si << endl;
        // cout << "Again, value of si = " << this->si << endl;
    }
};

int X::si = 77;           // Initialize static data member
```

```

int main() {
    X xobj;
    xobj.set_i(11);
    xobj.print_i();

    // static data members and functions belong to the class and
    // can be accessed without using an object of class X
    X::print_si();
    X::set_si(22);
    X::print_si();
}

```

The following is the output of the above example:

```

Value of i = 11
Again, value of i = 11
Value of si = 77
Value of si = 22

```

The compiler would not allow the member access operation `this->si` in function `A::print_si()` because this member function has been declared as static, and therefore does not have a **this** pointer.

You can call a static member function using the **this** pointer of a nonstatic member function. In the following example, the nonstatic member function `printall()` calls the static member function `f()` using the **this** pointer:

CCNX11H

```

#include <iostream>
using namespace std;

class C {
    static void f() {
        cout << "Here is i: " << i << endl;
    }
    static int i;
    int j;
public:
    C(int firstj): j(firstj) { }
    void printall();
};

void C::printall() {
    cout << "Here is j: " << this->j << endl;
    this->f();
}

int C::i = 3;

int main() {
    C obj_C(0);
    obj_C.printall();
}

```

The following is the output of the above example:

```

Here is j: 0
Here is i: 3

```

A static member function cannot be declared with the keywords **virtual**, **const**, **volatile**, or **const volatile**.

Static Members

A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared. Suppose a static member function `f()` is a member of class `X`. The static member function `f()` cannot access the nonstatic members `X` or the nonstatic members of a base class of `X`.

RELATED REFERENCES

- “The `this` Pointer” on page 300

Member Access

C++ *Member access* determines if a class member is accessible in an expression or declaration. Suppose `x` is a member of class `A`. Class member `x` can be one of the following:

- **public**: `x` can be used anywhere without the access restrictions defined by `private` or `protected`.
- **private**: `x` can be used only by the members and friends of class `A`.
- **protected**: `x` can be used only by the members and friends of class `A`, and the members and friends of classes derived from class `A`.

Members of classes declared with the keyword **class** are private by default. Members of classes declared with the keyword **struct** or **union** are public by default.

To control the access of a class member, you use one of the *access specifiers* **public**, **private**, or **protected** as a label in a class member list. The following example demonstrates these access specifiers:

```
struct A {
    friend class C;
private:
    int a;
public:
    int b;
protected:
    int c;
};

struct B : A {
    void f() {
        // a = 1;
        b = 2;
        c = 3;
    }
};

struct C {
    void f(A x) {
        x.a = 4;
        x.b = 5;
        x.c = 6;
    }
};

int main() {
    A y;
    // y.a = 7;
    y.b = 8;
    // y.c = 9;

    B z;
```

```
// z.a = 10;
   z.b = 11;
// z.c = 12;
}
```

The following table lists the access of data members `A::a`, `A::b`, and `A::c` in various scopes of the above example:

Scope	<code>A::a</code>	<code>A::b</code>	<code>A::c</code>
function <code>B::f()</code>	No access. Member <code>A::a</code> is private.	Access. Member <code>A::b</code> is public.	Access. Class <code>B</code> inherits from <code>A</code> .
function <code>C::f()</code>	Access. Class <code>C</code> is a friend of <code>A</code> .	Access. Member <code>A::b</code> is public.	Access. Class <code>C</code> is a friend of <code>A</code> .
object <code>y</code> in <code>main()</code>	No access. Member <code>y.a</code> is private.	Access. Member <code>y.a</code> is public.	No access. Member <code>y.c</code> is protected.
object <code>z</code> in <code>main()</code>	No access. Member <code>z.a</code> is private.	Access. Member <code>z.a</code> is public.	No access. Member <code>z.c</code> is protected.

An access specifier specifies the accessibility of members that follow it until the next access specifier or until the end of the class definition. You can use any number of access specifiers in any order. If you later define a class member within its class definition, its access specification must be the same as its declaration. The following example demonstrates this:

```
class A {
    class B;
    public:
        class B { };
};
```

The compiler will not allow the definition of class `B` because this class has already been declared as private.

A class member has the same access control regardless whether it has been defined within its class or outside its class.

Access control applies to names. In particular, if you add access control to a typedef name, it affects only the typedef name. The following example demonstrates this:

```
class A {
    class B { };
    public:
        typedef B C;
};

int main() {
    A::C x;
    // A::B y;
}
```

The compiler will allow the declaration `A::C x` because the typedef name `A::C` is public. The compiler would not allow the declaration `A::B y` because `A::B` is private.

Note that accessibility and visibility are independent. Visibility is based on the scoping rules of C++. A class member can be visible and inaccessible at the same time.

RELATED REFERENCES

- “Scope” on page 1
- “Class Member Lists” on page 293
- “Inherited Member Access” on page 320

Friends

C++ A friend of a class *X* is a function or class that is not a member of *X*, but is granted the same access to *X* as the members of *X*. Functions declared with the **friend** specifier in a class member list are called *friend functions* of that class. Classes declared with the **friend** specifier in the member list of another class are called *friend classes* of that class.

A class *Y* must be defined before any member of *Y* can be declared a friend of another class.

In the following example, the friend function `print` is a member of class *Y* and accesses the private data members `a` and `b` of class *X*.

CCNX11I

```
#include <iostream>
using namespace std;

class X;

class Y {
public:
    void print(X& x);
};

class X {
    int a, b;
    friend void Y::print(X& x);
public:
    X() : a(1), b(2) { }
};

void Y::print(X& x) {
    cout << "a is " << x.a << endl;
    cout << "b is " << x.b << endl;
}

int main() {
    X xobj;
    Y yobj;
    yobj.print(xobj);
}
```

The following is the output of the above example:

```
a is 1
b is 2
```

You can declare an entire class as a friend. Suppose class *F* is a friend of class *A*. This means that every member function and static data member definition of class *F* has access to class *A*.

In the following example, the friend class *F* has a member function `print` that accesses the private data members `a` and `b` of class *X* and performs the same task

as the friend function `print` in the above example. Any other members declared in class `F` also have access to all members of class `X`:

CCNX11J

```
#include <iostream>
using namespace std;

class X {
    int a, b;
    friend class F;
public:
    X() : a(1), b(2) { }
};

class F {
public:
    void print(X& x) {
        cout << "a is " << x.a << endl;
        cout << "b is " << x.b << endl;
    }
};

int main() {
    X xobj;
    F fobj;
    fobj.print(xobj);
}
```

The following is the output of the above example:

```
a is 1
b is 2
```

You must use an elaborated type specifier when you declare a class as a friend. The following example demonstrates this:

```
class F;
class G;
class X {
    friend class F;
    friend G;
};
```

VAC++ The VisualAge C++ compiler will warn you that the friend declaration of `G` must be an elaborated class name.

You cannot define a class in a friend declaration. For example, the compiler will not allow the following:

```
class F;
class X {
    friend class F { };
};
```

However, you can define a function in a friend declaration. The class must be a non-local class, function, the function name must be unqualified, and the function has namespace scope. The following example demonstrates this:

```
class A {
    void g();
};

void z() {
    class B {
    // friend void f() { };
    }
```

Friends

```
};  
}  
  
class C {  
    // friend void A::g() { }  
    friend void h() { }  
};
```

The compiler would not allow the function definition of `f()` or `g()`. The compiler will allow the definition of `h()`.

You cannot declare a friend with a storage class specifier.

RELATED REFERENCES

- “Member Access” on page 308

Friend Scope

> C++ The name of a friend function or class first introduced in a friend declaration is not in the scope of the class granting friendship (also called the *enclosing class*) and is not a member of the class granting friendship.

The name of a function first introduced in a friend declaration is in the scope of the first nonclass scope that contains the enclosing class. The body of a function provided in a friend declaration is handled in the same way as a member function defined within a class. Processing of the definition does not start until the end of the outermost enclosing class. In addition, unqualified names in the body of the function definition are searched for starting from the class containing the function definition.

A class that is first declared in a friend declaration is equivalent to an **extern** declaration. For example:

```
class B {};  
class A  
{  
    friend class B; // global class B is a friend of A  
};
```

If the name of a friend class has been introduced before the friend declaration, the compiler searches for a class name that matches the name of the friend class beginning at the scope of the friend declaration. If the declaration of a nested class is followed by the declaration of a friend class with the same name, the nested class is a friend of the enclosing class.

The scope of a friend class name is the first nonclass enclosing scope. For example:

```
class A {  
    class B { // arbitrary nested class definitions  
        friend class C;  
    };  
};
```

is equivalent to:

```
class C;  
class A {  
    class B { // arbitrary nested class definitions  
        friend class C;  
    };  
};
```

If the friend function is a member of another class, you need to use the scope resolution operator (::). For example:

```
class A {
public:
    int f() { }
};

class B {
    friend int A::f();
};
```

Friends of a base class are not inherited by any classes derived from that base class. The following example demonstrates this:

```
class A {
    friend class B;
    int a;
};

class B { };

class C : public B {
    void f(A* p) {
        // p->a = 2;
    }
};
```

The compiler would not allow the statement `p->a = 2` because class `C` is not a friend of class `A`, although `C` inherits from a friend of `A`.

Friendship is not transitive. The following example demonstrates this:

```
class A {
    friend class B;
    int a;
};

class B {
    friend class C;
};

class C {
    void f(A* p) {
        // p->a = 2;
    }
};
```

The compiler would not allow the statement `p->a = 2` because class `C` is not a friend of class `A`, although `C` is a friend of a friend of `A`.

If you declare a friend in a local class, and the friend's name is unqualified, the compiler will look for the name only within the innermost enclosing nonclass scope. You must declare a function before declaring it as a friend of a local scope. You do not have to do so with classes. However, a declaration of a friend class will hide a class in an enclosing scope with the same name. The following example demonstrates this:

```
class X { };
void a();

void f() {
    class Y { };
    void b();
    class A {
        friend class X;
```

Friends

```
        friend class Y;
        friend class Z;
    //    friend void a();
        friend void b();
    //    friend void c();
    };
    ::X moocow;
    // X moocow2;
}
```

In the above example, the compiler will allow the following statements:

- `friend class X`: This statement does not declare `::X` as a friend of `A`, but the local class `X` as a friend, even though this class is not otherwise declared.
- `friend class Y`: Local class `Y` has been declared in the scope of `f()`.
- `friend class Z`: This statement declares the local class `Z` as a friend of `A` even though `Z` is not otherwise declared.
- `friend void b()`: Function `b()` has been declared in the scope of `f()`.
- `::X moocow`: This declaration creates an object of the nonlocal class `::X`.

The compiler would not allow the following statements:

- `friend void a()`: This statement does not consider function `a()` declared in namespace scope. Since function `a()` has not been declared in the scope of `f()`, the compiler would not allow this statement.
- `friend void c()`: Since function `c()` has not been declared in the scope of `f()`, the compiler would not allow this statement.
- `X moocow2`: This declaration tries to create an object of the local class `X`, not the nonlocal class `::X`. Since local class `X` has not been defined, the compiler would not allow this statement.

RELATED REFERENCES

- “Scope of Class Names” on page 287
- “Nested Classes” on page 288
- “Dot Operator `.`” on page 107
- “Derivation” on page 317
- “External Linkage” on page 6

Friend Access

C++ A friend of a class can access the private and protected members of that class. Normally, you can only access the private members of a class through member functions of that class, and you can only access the protected members of a class through member functions of a class or classes derived from that class.

Friend declarations are not affected by access specifiers.

RELATED REFERENCES

- “Friends” on page 310
- “Member Access” on page 308

Chapter 14. Inheritance

C++ *Inheritance* is a mechanism of reusing and extending existing classes without modifying them.

Inheritance is almost like embedding an object into a class. Suppose that you declare an object *x* of class *A* in the class definition of *B*. As a result, class *B* will have access to all the public data members and member functions of class *A*. However, in class *B*, you have to access the data members and member functions of class *A* through object *x*. The following example demonstrates this:

```
#include <iostream>
using namespace std;

class A {
    int data;
public:
    void f(int arg) { data = arg; }
    int g() { return data; }
};

class B {
public:
    A x;
};

int main() {
    B obj;
    obj.x.f(20);
    cout << obj.x.g() << endl;
    // cout << obj.g() << endl;
}
```

In the main function, object *obj* accesses function *A::f()* through its data member *B::x* with the statement *obj.x.f(20)*. Object *obj* accesses *A::g()* in a similar manner with the statement *obj.x.g()*. The compiler would not allow the statement *obj.g()* because *g()* is a member function of class *A*, not class *B*.

The inheritance mechanism lets you use a statement like *obj.g()* in the above example. In order for that statement to be legal, *g()* must be a member function of class *B*.

Inheritance lets you include the names and definitions of another class's members as part of a new class. The class whose members you want to include in your new class is called a *base class*. Your new class is *derived* from the base class. Your new class will contain a *subobject* of the type of the base class. The following example is the same as the previous example except it uses the inheritance mechanism to give class *B* access to the members of class *A*:

```
#include <iostream>
using namespace std;

class A {
    int data;
public:
    void f(int arg) { data = arg; }
    int g() { return data; }
};

class B : public A { };
```

```
int main() {
    B obj;
    obj.f(20);
    cout << obj.g() << endl;
}
```

Class A is a base class of class B. The names and definitions of the members of class A are included in the definition of class B; class B inherits the members of class A. Class B is derived from class A. Class B contains a subobject of type A.

You can also add new data members and member functions to the derived class. You can modify the implementation of existing member functions or data by overriding base class member functions or data in the newly derived class.

You may derive classes from other derived classes, thereby creating another level of inheritance. The following example demonstrates this:

```
struct A { };
struct B : A { };
struct C : B { };
```

Class B is a derived class of A, but is also a base class of C. The number of levels of inheritance is only limited by resources.

Multiple inheritance allows you to create a derived class that inherits properties from more than one base class. Because a derived class inherits members from all its base classes, ambiguities can result. For example, if two base classes have a member with the same name, the derived class cannot implicitly differentiate between the two members. Note that, when you are using multiple inheritance, the access to names of base classes may be ambiguous.

A *direct base class* is a base class that appears directly as a base specifier in the declaration of its derived class.

An *indirect base class* is a base class that does not appear directly in the declaration of the derived class but is available to the derived class through one of its base classes. For a given class, all base classes that are not direct base classes are indirect base classes. The following example demonstrates direct and indirect base classes:

```
class A {
public:
    int x;
};
class B : public A {
public:
    int y;
};
class C : public B { };
```

Class B is a direct base class of C. Class A is a direct base class of B. Class A is an indirect base class of C. (Class C has x and y as its data members.)

Polymorphic functions are functions that can be applied to objects of more than one type. In C++, polymorphic functions are implemented in two ways:

- Overloaded functions are statically bound at compile time.
- C++ provides virtual functions. A *virtual function* is a function that can be called for a number of different user-defined types that are related through derivation. Virtual functions are bound dynamically at run time.

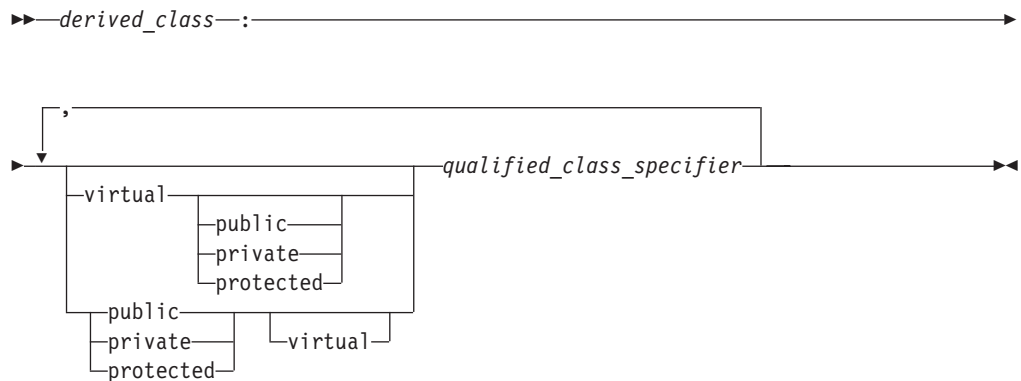
RELATED REFERENCES

- “Multiple Inheritance” on page 327
- “Overloading Functions” on page 269
- “Virtual Functions” on page 333
- “Chapter 7. Functions” on page 153
- “Chapter 12. Classes” on page 283
- “Chapter 13. Class Members and Friends” on page 293

Derivation

► **C++** Inheritance is implemented in C++ through the mechanism of derivation. Derivation allows you to derive a class, called a *derived class*, from another class, called a *base class*.

Syntax — Derived Class Derivation



In the declaration of a derived class, you list the base classes of the derived class. The derived class inherits its members from these base classes.

The *qualified_class_specifier* must be a class that has been previously declared in a class declaration.

An *access specifier* is one of **public**, **private**, or **protected**.

The **virtual** keyword can be used to declare virtual base classes.

The following example shows the declaration of the derived class D and the base classes V, B1, and B2. The class B1 is both a base class and a derived class because it is derived from class V and is a base class for D:

```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 { /* ... */ };
class D : public B1, private B2 { /* ... */ };
```

Classes that are declared but not defined are not allowed in base lists.

For example:

```
class X;

// error
class Y: public X { };
```

Derivation

The compiler will not allow the declaration of class Y because X has not been defined.

When you derive a class, the derived class inherits class members of the base class. You can refer to inherited members (base class members) as if they were members of the derived class. For example:

CCNX14A

```
class Base {
public:
    int a,b;
};

class Derived : public Base {
public:
    int c;
};

int main() {
    Derived d;
    d.a = 1;    // Base::a
    d.b = 2;    // Base::b
    d.c = 3;    // Derived::c
}
```

The derived class can also add new class members and redefine existing base class members. In the above example, the two inherited members, a and b, of the derived class d, in addition to the derived class member c, are assigned values. If you redefine base class members in the derived class, you can still refer to the base class members by using the :: (scope resolution) operator. For example:

CCNX14B

```
#include <iostream>
using namespace std;

class Base {
public:
    char* name;
    void display() {
        cout << name << endl;
    }
};

class Derived: public Base {
public:
    char* name;
    void display() {
        cout << name << ", " << Base::name << endl;
    }
};

int main() {
    Derived d;
    d.name = "Derived Class";
    d.Base::name = "Base Class";

    // call Derived::display()
    d.display();

    // call Base::display()
    d.Base::display();
}
```


The following is the output of the above example:

```
Derived Class, Base Class
Base Class
```

You can manipulate a derived class object as if it were a base class object. You can use a pointer or a reference to a derived class object in place of a pointer or reference to its base class. For example, you can pass a pointer or reference to a derived class object D to a function expecting a pointer or reference to the base class of D. You do not need to use an explicit cast to achieve this; a standard conversion is performed. You can implicitly convert a pointer to a derived class to point to an accessible unambiguous base class. You can also implicitly convert a reference to a derived class to a reference to a base class.

The following example demonstrates a standard conversion from a pointer to a derived class to a pointer to a base class:

CCNX14C

```
#include <iostream>
using namespace std;

class Base {
public:
    char* name;
    void display() {
        cout << name << endl;
    }
};

class Derived: public Base {
public:
    char* name;
    void display() {
        cout << name << ", " << Base::name << endl;
    }
};

int main() {
    Derived d;
    d.name = "Derived Class";
    d.Base::name = "Base Class";

    Derived* dptr = &d;

    // standard conversion from Derived* to Base*
    Base* bptr = dptr;

    // call Base::display()
    bptr->display();
}
```

The following is the output of the above example:

```
Base Class
```

The statement `Base* bptr = dptr` converts a pointer of type `Derived` to a pointer of type `Base`.

The reverse case is not allowed. You cannot implicitly convert a pointer or a reference to a base class object to a pointer or reference to a derived class. For example, the compiler will not allow the following code if the classes `Base` and `Class` are defined as in the above example:

Derivation

```
int main() {
    Base b;
    b.name = "Base class";

    Derived* dptr = &b;
}
```

The compiler will not allow the statement `Derived* dptr = &b` because the statement is trying to implicitly convert a pointer of type `Base` to a pointer of type `Derived`.

If a member of a derived class and a member of a base class have the same name, the base class member is hidden in the derived class. If a member of a derived class has the same name as a base class, the base class name is hidden in the derived class.

RELATED REFERENCES

- “Virtual Base Classes” on page 328
- “Incomplete Class Declarations” on page 288
- “C++ Scope Resolution Operator `::`” on page 102
- “Member Access” on page 308
- “References” on page 92

Inherited Member Access

This section consists of a discussion of the classes that can access a protected nonstatic base class member and how to declare a derived class using an access specifier.

Protected Members

C++ A protected nonstatic base class member can be accessed by members and friends of any classes derived from that base class by using one of the following:

- A pointer to a directly or indirectly derived class
- A reference to a directly or indirectly derived class
- An object of a directly or indirectly derived class

If a class is derived privately from a base class, all protected base class members become private members of the derived class.

If you reference a protected nonstatic member `x` of a base class `A` in a friend or a member function of a derived class `B`, you must access `x` through a pointer to, reference to, or object of a class derived from `A`. However, if you are accessing `x` to create a pointer to member, you must qualify `x` with a nested name specifier that names the derived class `B`. The following example demonstrates this:

```
class A {
public:
protected:
    int i;
};

class B : public A {
    friend void f(A*, B*);
    void g(A*);
};
```

```

void f(A* pa, B* pb) {
    // pa->i = 1;
    pb->i = 2;

    // int A::* point_i = &A::i;
    int A::* point_i2 = &B::i;
}

void B::g(A* pa) {
    // pa->i = 1;
    i = 2;

    // int A::* point_i = &A::i;
    int A::* point_i2 = &B::i;
}

void h(A* pa, B* pb) {
    // pa->i = 1;
    // pb->i = 2;
}

int main() { }

```

Class A contains one protected data member, an integer `i`. Because B derives from A, the members of B have access to the protected member of A. Function `f()` is a friend of class B:

- The compiler would not allow `pa->i = 1` because `pa` is not a pointer to the derived class B.
- The compiler would not allow `int A::* point_i = &A::i` because `i` has not been qualified with the name of the derived class B.

Function `g()` is a member function of class B. The previous list of remarks about which statements the compiler would and would not allow apply for `g()` except for the following:

- The compiler allows `i = 2` because it is equivalent to `this->i = 2`.

Function `h()` cannot access any of the protected members of A because `h()` is neither a friend or a member of a derived class of A.

RELATED REFERENCES

- “References” on page 92
- “Objects” on page 34

Access Control of Base Class Members

C++ When you declare a derived class, an access specifier can precede each base class in the base list of the derived class. This does not alter the access attributes of the individual members of a base class as seen by the base class, but allows the derived class to restrict the access control of the members of a base class.

You can derive classes using any of the three access specifiers:

- In a **public** base class, public and protected members of the base class remain public and protected members of the derived class.
- In a **protected** base class, public and protected members of the base class are protected members of the derived class.
- In a **private** base class, public and protected members of the base class become private members of the derived class.

Inherited Member Access

In all cases, private members of the base class remain private. Private members of the base class cannot be used by the derived class unless friend declarations within the base class explicitly grant access to them.

In the following example, class `d` is derived publicly from class `b`. Class `b` is declared a public base class by this declaration.

```
class b { };
class d : public b // public derivation
{ };
```

You can use both a structure and a class as base classes in the base list of a derived class declaration:

- If the derived class is declared with the keyword **class**, the default access specifier in its base list specifiers is **private**.
- If the derived class is declared with the keyword **struct**, the default access specifier in its base list specifiers is **public**.

In the following example, private derivation is used by default because no access specifier is used in the base list and the derived class is declared with the keyword **class**:

```
struct B
{ };
class D : B // private derivation
{ };
```

Members and friends of a class can implicitly convert a pointer to an object of that class to a pointer to either:

- A direct private base class
- A protected base class (either direct or indirect)

RELATED REFERENCES

- “Member Access” on page 308
- “Structures” on page 51
- “Chapter 12. Classes” on page 283
- “Chapter 13. Class Members and Friends” on page 293
- “Friends” on page 310

The using Declaration and Class Members

C++ A using declaration in a definition of a class `A` allows you to introduce a *name* of a data member or member function from a base class of `A` into the scope of `A`.

You would need a using declaration in a class definition if you want to create a set of overload a member functions from base and derived classes, or you want to change the access of a class member.

Syntax — using Declaration

```
→ using [typename] [::] nested_name_specifier unqualified_id ; →
      [::] unqualified_id ;
```

A using declaration in a class `A` may name one of the following:

- A member of a base class of A
- A member of an anonymous union that is a member of a base class of A
- An enumerator for an enumeration type that is a member of a base class of A

The following example demonstrates this:

```
struct Z {
    int g();
};

struct A {
    void f();
    enum E { e };
    union { int u; };
};

struct B : A {
    using A::f;
    using A::e;
    using A::u;
    // using Z::g;
};
```

The compiler would not allow the using declaration using `Z::g` because `Z` is not a base class of `A`.

A using declaration cannot name a template. For example, the compiler will not allow the following:

```
struct A {
    template<class T> void f(T);
};

struct B : A {
    using A::f<int>;
};
```

Every instance of the name mentioned in a using declaration must be accessible. The following example demonstrates this:

```
struct A {
private:
    void f(int);
public:
    int f();
protected:
    void g();
};

struct B : A {
    // using A::f;
    using A::g;
};
```

The compiler would not allow the using declaration using `A::f` because `void A::f(int)` is not accessible from `B` even though `int A::f()` is accessible.

RELATED REFERENCES

- “Scope of Class Names” on page 287
- “Overloading Member Functions from Base and Derived Classes” on page 324
- “Changing the Access of a Class Member” on page 325
- “The using Declaration and Namespaces” on page 267

Overloading Member Functions from Base and Derived Classes

► C++ A member function named `f` in a class `A` will hide all other members named `f` in the base classes of `A`, regardless of return types or arguments. The following example demonstrates this:

```
struct A {
    void f() { }
};

struct B : A {
    void f(int) { }
};

int main() {
    B obj_B;
    obj_B.f(3);
    // obj_B.f();
}
```

The compiler would not allow the function call `obj_B.f()` because the declaration of `void B::f(int)` has hidden `A::f()`.

To overload, rather than hide, a function of a base class `A` in a derived class `B`, you introduce the name of the function into the scope of `B` with a `using` declaration. The following example is the same as the previous example except for the `using` declaration using `A::f`:

```
struct A {
    void f() { }
};

struct B : A {
    using A::f;
    void f(int) { }
};

int main() {
    B obj_B;
    obj_B.f(3);
    obj_B.f();
}
```

Because of the `using` declaration in class `B`, the name `f` is overloaded with two functions. The compiler will now allow the function call `obj_B.f()`.

You can overload virtual functions in the same way. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "void A::f()" << endl; }
    virtual void f(int) { cout << "void A::f(int)" << endl; }
};

struct B : A {
    using A::f;
    void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
    B obj_B;
```

```

    B* pb = &obj_B;
    pb->f(3);
    pb->f();
}

```

The following is the output of the above example:

```

void B::f(int)
void A::f()

```

Suppose that you introduce a function `f` from a base class `A` a derived class `B` with a using declaration, and there exists a function named `B::f` that has the same parameter types as `A::f`. Function `B::f` will hide, rather than conflict with, function `A::f`. The following example demonstrates this:

```

#include <iostream>
using namespace std;

struct A {
    void f() { }
    void f(int) { cout << "void A::f(int)" << endl; }
};

struct B : A {
    using A::f;
    void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
    B obj_B;
    obj_B.f(3);
}

```

The following is the output of the above example:

```

void B::f(int)

```

RELATED REFERENCES

- “Chapter 11. Overloading” on page 269
- “Name Hiding” on page 4
- “The using Declaration and Class Members” on page 322

Changing the Access of a Class Member

C++ Suppose class `B` is a direct base class of class `A`. To restrict access of class `B` to the members of class `A`, derive `B` from `A` using either the access specifiers **protected** or **private**.

To increase the access of a member `x` of class `A` inherited from class `B`, use a using declaration. You cannot restrict the access to `x` with a using declaration. You may increase the access of the following members:

- A member inherited as **private**. (You cannot increase the access of a member declared as **private** because a using declaration must have access to the member’s name.)
- A member either inherited or declared as **protected**

The following example demonstrates this:

```

struct A {
protected:
    int y;
public:
    int z;
}

```

Inherited Member Access

```
};

struct B : private A { };

struct C : private A {
public:
    using A::y;
    using A::z;
};

struct D : private A {
protected:
    using A::y;
    using A::z;
};

struct E : D {
    void f() {
        y = 1;
        z = 2;
    }
};

struct F : A {
public:
    using A::y;
private:
    using A::z;
};

int main() {
    B obj_B;
    // obj_B.y = 3;
    // obj_B.z = 4;

    C obj_C;
    obj_C.y = 5;
    obj_C.z = 6;

    D obj_D;
    // obj_D.y = 7;
    // obj_D.z = 8;

    F obj_F;
    obj_F.y = 9;
    obj_F.z = 10;
}
```

The compiler would not allow the following assignments from the above example:

- `obj_B.y = 3` and `obj_B.z = 4`: Members `y` and `z` have been inherited as **private**.
- `obj_D.y = 7` and `obj_D.z = 8`: Members `y` and `z` have been inherited as **private**, but their access have been changed to **protected**.

The compiler allows the following statements from the above example:

- `y = 1` and `z = 2` in `D::f()`: Members `y` and `z` have been inherited as **private**, but their access have been changed to **protected**.
- `obj_C.y = 5` and `obj_C.z = 6`: Members `y` and `z` have been inherited as **private**, but their access have been changed to **public**.
- `obj_F.y = 9`: The access of member `y` has been changed from **protected** to **public**.
- `obj_F.z = 10`: The access of member `z` is still **public**. The **private** using declaration using `A::z` has no effect on the access of `z`.

RELATED REFERENCES

- “Member Access” on page 308
- “Protected Members” on page 320
- “Access Control of Base Class Members” on page 321
- “The using Declaration and Class Members” on page 322

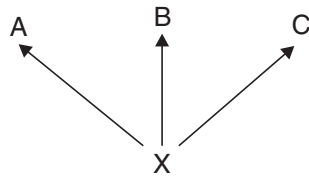
Multiple Inheritance

C++ You can derive a class from any number of base classes. Deriving a class from more than one direct base class is called *multiple inheritance*.

In the following example, classes A, B, and C are direct base classes for the derived class X:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```

The following *inheritance graph* describes the inheritance relationships of the above example. An arrow points to the direct base class of the class at the tail of the arrow:

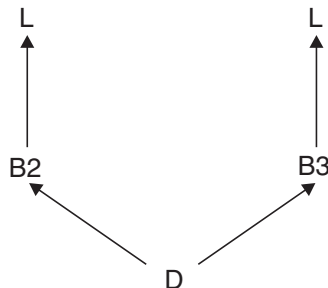


The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.

A direct base class cannot appear in the base list of a derived class more than once:

```
class B1 { /* ... */ }; // direct base class
class D : public B1, private B1 { /* ... */ }; // error
```

However, a derived class can inherit an indirect base class more than once, as shown in the following example:



```
class L { /* ... */ }; // indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
class D : public B2, public B3 { /* ... */ }; // valid
```

In the above example, class D inherits the indirect base class L once through class B2 and once through class B3. However, this may lead to ambiguities because two

Multiple Inheritance

subobjects of class L exist, and both are accessible through class D. You can avoid this ambiguity by referring to class L using a qualified class name. For example:

B2::L

or

B3::L.

You can also avoid this ambiguity by using the base specifier **virtual** to declare a base class.

RELATED REFERENCES

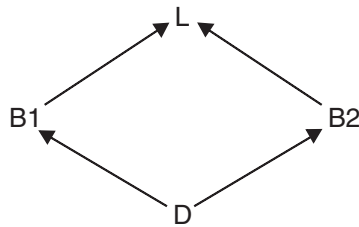
- “Virtual Base Classes”

Virtual Base Classes

> C++ Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as *virtual* to ensure that B and C share the same subobject of A.

In the following example, an object of class D has two distinct subobjects of class L, one through class B1 and another through class B2. You can use the keyword **virtual** in front of the base class specifiers in the *base lists* of classes B1 and B2 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.

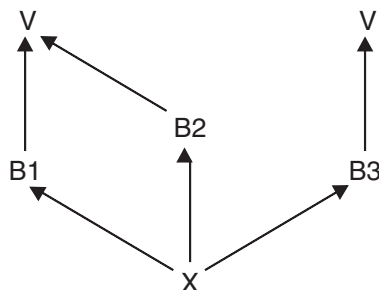
For example:



```
class L { /* ... */ }; // indirect base class
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ }; // valid
```

Using the keyword **virtual** in this example ensures that an object of class D inherits only one subobject of class L.

A derived class can have both virtual and nonvirtual base classes. For example:



```

class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 : virtual public V { /* ... */ };
class B3 : public V { /* ... */ };
class D : public B1, public B2, public B3 { /* ... */ };

```

In the above example, class D has two subobjects of class V, one that is shared by classes B1 and B2 and one through class B3.

RELATED REFERENCES

- “Derivation” on page 317

Multiple Access

C++ In an inheritance graph containing virtual base classes, a name that can be reached through more than one path is accessed through the path that gives the most access.

For example:

```

class L {
public:
    void f();
};

class B1 : private virtual L { };

class B2 : public virtual L { };

class D : public B1, public B2 {
public:
    void f() {
        // L::f() is accessed through B2
        // and is public
        L::f();
    }
};

```

In the above example, the function `f()` is accessed through class B2. Because class B2 is inherited publicly and class B1 is inherited privately, class B2 offers more access.

RELATED REFERENCES

- “Member Access” on page 308
- “Protected Members” on page 320
- “Access Control of Base Class Members” on page 321

Ambiguous Base Classes

C++ When you derive classes, ambiguities can result if base and derived classes have members with the same names. Access to a base class member is ambiguous if you use a name or qualified name that does not refer to a unique function or object. The declaration of a member with an ambiguous name in a derived class is not an error. The ambiguity is only flagged as an error if you use the ambiguous member name.

For example, suppose that two classes named A and B both have a member named `x`, and a class named C inherits from both A and B. An attempt to access `x` from

Multiple Inheritance

class C would be ambiguous. You can resolve ambiguity by qualifying a member with its class name using the scope resolution (::) operator.

CCNX14G

```
class B1 {
public:
    int i;
    int j;
    void g(int) { }
};

class B2 {
public:
    int j;
    void g() { }
};

class D : public B1, public B2 {
public:
    int i;
};

int main() {
    D dobj;
    D *dptr = &dobj;
    dptr->i = 5;
    // dptr->j = 10;
    dptr->B1::j = 10;
    // dobj.g();
    dobj.B2::g();
}
```

The statement `dptr->j = 10` is ambiguous because the name `j` appears both in `B1` and `B2`. The statement `dobj.g()` is ambiguous because the name `g` appears both in `B1` and `B2`, even though `B1::g(int)` and `B2::g()` have different parameters.

The compiler checks for ambiguities at compile time. Because ambiguity checking occurs before access control or type checking, ambiguities may result even if only one of several members with the same name is accessible from the derived class.

Name Hiding

Suppose two subobjects named `A` and `B` both have a member name `x`. The member name `x` of subobject `B` *hides* the member name `x` of subobject `A` if `A` is a base class of `B`. The following example demonstrates this:

```
struct A {
    int x;
};

struct B: A {
    int x;
};

struct C: A, B {
    void f() { x = 0; }
};

int main() {
    C i;
    i.f();
}
```

The assignment `x = 0` in function `C::f()` is not ambiguous because the declaration `B::x` has hidden `A::x`. However, the compiler will warn you that deriving `C` from `A` is redundant because you already have access to the subobject `A` through `B`.

A base class declaration can be hidden along one path in the inheritance graph and not hidden along another path. The following example demonstrates this:

```
struct A { int x; };
struct B { int y; };
struct C: A, virtual B { };
struct D: A, virtual B {
    int x;
    int y;
};
struct E: C, D { };

int main() {
    E e;
    // e.x = 1;
    e.y = 2;
}
```

The assignment `e.x = 1` is ambiguous. The declaration `D::x` hides `A::x` along the path `D::A::x`, but it does not hide `A::x` along the path `D::B::A::x`. Therefore the variable `x` could refer to either `D::x` or `A::x`. The assignment `e.y = 2` is not ambiguous. The declaration `D::y` hides `B::y` along both paths `D::B::y` and `C::B::y` because `B` is a virtual base class.

Ambiguity and using Declarations

Suppose you have a class named `C` that inherits from a class named `A`, and `x` is a member name of `A`. If you use a using declaration to declare `A::x` in `C`, then `x` is also a member of `C`; `C::x` does not hide `A::x`. Therefore using declarations cannot resolve ambiguities due to inherited members. The following example demonstrates this:

```
struct A {
    int x;
};

struct B: A { };

struct C: A {
    using A::x;
};

struct D: B, C {
    void f() { x = 0; }
};

int main() {
    D i;
    i.f();
}
```

The compiler will not allow the assignment `x = 0` in function `D::f()` because it is ambiguous. The compiler can find `x` in two ways: as `B::x` or as `C::x`.

Unambiguous Class Members

The compiler can unambiguously find static members, nested types, and enumerators defined in a base class `A` regardless of the number of subobjects of type `A` an object has. The following example demonstrates this:

Multiple Inheritance

```
struct A {
    int x;
    static int s;
    typedef A* Pointer_A;
    enum { e };
};

int A::s;

struct B: A { };

struct C: A { };

struct D: B, C {
    void f() {
        s = 1;
        Pointer_A pa;
        int i = e;
        // x = 1;
    }
};

int main() {
    D i;
    i.f();
}
```

The compiler allows the assignment `s = 1`, the declaration `Pointer_A pa`, and the statement `int i = e`. There is only one static variable `s`, only one typedef `Pointer_A`, and only one enumerator `e`. The compiler would not allow the assignment `x = 1` because `x` can be reached either from class `B` or class `C`.

Pointer Conversions

Conversions (either implicit or explicit) from a derived class pointer or reference to a base class pointer or reference must refer unambiguously to the same accessible base class object. (An *accessible base class* is a publicly derived base class that is neither hidden nor ambiguous in the inheritance hierarchy.) For example:

```
class W { /* ... */ };
class X : public W { /* ... */ };
class Y : public W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int main ()
{
    Z z;
    X* xptr = &z;      // valid
    Y* yptr = &z;      // valid
    W* wptr = &z;      // error, ambiguous reference to class W
                      // X's W or Y's W ?
}
```

You can use virtual base classes to avoid ambiguous reference. For example:

```
class W { /* ... */ };
class X : public virtual W { /* ... */ };
class Y : public virtual W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int main ()
{
    Z z;
    X* xptr = &z;      // valid
    Y* yptr = &z;      // valid
    W* wptr = &z;      // valid, W is virtual therefore only one
                      // W subobject exists
}
```

Overload Resolution

Overload resolution takes place *after* the compiler unambiguously finds a given function name. The following example demonstrates this:

```
struct A {
    int f() { return 1; }
};

struct B {
    int f(int arg) { return arg; }
};

struct C: A, B {
    int g() { return f(); }
};
```

The compiler will not allow the function call to `f()` in `C::g()` because the name `f` has been declared both in `A` and `B`. The compiler detects the ambiguity error before overload resolution can select the base match `A::f()`.

RELATED REFERENCES

- “C++ Scope Resolution Operator `::`” on page 102
- “Virtual Base Classes” on page 328

Virtual Functions

C++ By default, C++ matches a function call with the correct function definition at compile time. This is called *static binding*. You can specify that the compiler match a function call with the correct function definition at run time; this is called *dynamic binding*. You declare a function with the keyword **virtual** if you want the compiler to use dynamic binding for that specific function.

The following examples demonstrate the differences between static and dynamic binding. The first example demonstrates static binding:

```
#include <iostream>
using namespace std;

struct A {
    void f() { cout << "Class A" << endl; }
};

struct B: A {
    void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
    arg.f();
}

int main() {
    B x;
    g(x);
}
```

The following is the output of the above example:

```
Class A
```

When function `g()` is called, function `A::f()` is called, although the argument refers to an object of type `B`. At compile time, the compiler knows only that the argument

Virtual Functions

of function `g()` will be a reference to an object derived from `A`; it cannot determine whether the argument will be a reference to an object of type `A` or type `B`. However, this can be determined at run time. The following example is the same as the previous example, except that `A::f()` is declared with the **virtual** keyword:

```
#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
    void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
    arg.f();
}

int main() {
    B x;
    g(x);
}
```

The following is the output of the above example:

```
Class B
```

The **virtual** keyword indicates to the compiler that it should choose the appropriate definition of `f()` not by the type of reference, but by the type of object that the reference refers to.

Therefore, a *virtual function* is a member function you may redefine for other derived classes, and can ensure that the compiler will call the redefined virtual function for an object of the corresponding derived class, even if you call that function with a pointer or reference to a base class of the object.

A class that declares or inherits a virtual function is called a *polymorphic class*

You redefine a virtual member function, like any member function, in any derived class. Suppose you declare a virtual function named `f` in a class `A`, and you derive directly or indirectly from `A` a class named `B`. If you declare a function named `f` in class `B` with the same name and same parameter list as `A::f`, then `B::f` is also virtual (regardless whether or not you declare `B::f` with the **virtual** keyword) and it *overrides* `A::f`. However, if the parameter lists of `A::f` and `B::f` are different, `A::f` and `B::f` are considered different, `B::f` does not override `A::f`, and `B::f` is not virtual (unless you have declared it with the **virtual** keyword). Instead `B::f` *hides* `A::f`. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
    void f(int) { cout << "Class B" << endl; }
};

struct C: B {
    void f() { cout << "Class C" << endl; }
};
```



```

int main() {
    B b; C c;
    A* pa1 = &b;
    A* pa2 = &c;
    //    b.f();
    pa1->f();
    pa2->f();
}

```

The following is the output of the above example:

```

Class A
Class C

```

The function `B::f` is not virtual. It hides `A::f`. Thus the compiler will not allow the function call `b.f()`. The function `C::f` is virtual; it overrides `A::f` even though `A::f` is not visible in `C`.

If you declare a base class destructor as virtual, a derived class destructor will override that base class destructor, even though destructors are not inherited.

The return type of an overriding virtual function may differ from the return type of the overridden virtual function. This overriding function would then be called a *covariant virtual function*. Suppose that `B::f` overrides the virtual function `A::f`. The return types of `A::f` and `B::f` may differ if all the following conditions are met:

- The function `B::f` returns a reference or pointer to a class of type `T`, and `A::f` returns a pointer or a reference to an unambiguous direct or indirect base class of `T`.
- The `const` or `volatile` qualification of the pointer or reference returned by `B::f` has the same or less `const` or `volatile` qualification of the pointer or reference returned by `A::f`.
- The return type of `B::f` must be complete at the point of declaration of `B::f`, or it can be of type `B`.

The following example demonstrates this:

```

#include <iostream>
using namespace std;

struct A { };

class B : private A {
    friend class D;
    friend class F;
};

A global_A;
B global_B;

struct C {
    virtual A* f() {
        cout << "A* C::f()" << endl;
        return &global_A;
    }
};

struct D : C {
    B* f() {
        cout << "B* D::f()" << endl;
        return &global_B;
    }
};

```

Virtual Functions

```
struct E;

struct F : C {

    // Error:
    // E is incomplete
    // E* f();
};

struct G : C {

    // Error:
    // A is an inaccessible base class of B
    // B* f();
};

int main() {
    D d;
    C* cp = &d;
    D* dp = &d;

    A* ap = cp->f();
    B* bp = dp->f();
};
```

The following is the output of the above example:

```
B* D::f()
B* D::f()
```

The statement `A* ap = cp->f()` calls `D::f()` and converts the pointer returned to type `A*`. The statement `B* bp = dp->f()` calls `D::f()` as well but does not convert the pointer returned; the type returned is `B*`. The compiler would not allow the declaration of the virtual function `F::f()` because `E` is not a complete class. The compiler would not allow the declaration of the virtual function `G::f()` because class `A` is not an accessible base class of `B` (unlike friend classes `D` and `F`, the definition of `B` does not give access to its members for class `G`).

A virtual function cannot be global or static because, by definition, a virtual function is a member function of a base class and relies on a specific object to determine which implementation of the function is called. You can declare a virtual function to be a friend of another class.

If a function is declared virtual in its base class, you can still access it directly using the scope resolution (`::`) operator. In this case, the virtual function call mechanism is suppressed and the function implementation defined in the base class is used. In addition, if you do not override a virtual member function in a derived class, a call to that function uses the function implementation defined in the base class.

A virtual function must be one of the following:

- Defined
- Declared pure
- Defined and declared pure

A base class containing one or more pure virtual member functions is called an *abstract class*.

RELATED REFERENCES

- “Function Return Values” on page 171
- “Abstract Classes” on page 339
- “Friends” on page 310

- “C++ Scope Resolution Operator ::” on page 102

Ambiguous Virtual Function Calls

> C++ You cannot override one virtual function with two or more ambiguous virtual functions. This can happen in a derived class that inherits from two nonvirtual bases that are derived from a virtual base class.

For example:

```
class V {
public:
    virtual void f() { }
};

class A : virtual public V {
    void f() { }
};

class B : virtual public V {
    void f() { }
};

// Error:
// Both A::f() and B::f() try to override V::f()
class D : public A, public B { };

int main() {
    D d;
    V* vptr = &d;

    // which f(), A::f() or B::f()?
    vptr->f();
}
```

The compiler will not allow the definition of class D. In class A, only A::f() will override V::f(). Similarly, in class B, only B::f() will override V::f(). However, in class D, both A::f() and B::f() will try to override V::f(). This attempt is not allowed because it is not possible to decide which function to call if a D object is referenced with a pointer to class V, as shown in the above example. Only one function can override a virtual function.

A special case occurs when the ambiguous overriding virtual functions come from separate instances of the same class type. In the following example, class D has two separate subobjects of class A:

```
#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "A::f()" << endl; };
};

struct B : A {
    void f() { cout << "B::f()" << endl; };
};

struct C : A {
    void f() { cout << "C::f()" << endl; };
};

struct D : B, C { };

int main() {
```

Virtual Functions

```
D d;  
  
B* bp = &d;  
A* ap = bp;  
D* dp = &d;  
  
ap->f();  
// dp->f();  
}
```

Class D has two occurrences of class A, one inherited from B, and another inherited from C. Therefore there are also two occurrences of the virtual function `A::f`. The statement `ap->f()` calls `D::B::f`. However the compiler would not allow the statement `dp->f()` because it could either call `D::B::f` or `D::C::f`.

Virtual Function Access

C++ The access for a virtual function is specified when it is declared. The access rules for a virtual function are not affected by the access rules for the function that later overrides the virtual function. In general, the access of the overriding member function is not known.

If a virtual function is called with a pointer or reference to a class object, the type of the class object is not used to determine the access of the virtual function. Instead, the type of the pointer or reference to the class object is used.

In the following example, when the function `f()` is called using a pointer having type `B*`, `bp` is used to determine the access to the function `f()`. Although the definition of `f()` defined in class D is executed, the access of the member function `f()` in class B is used. When the function `f()` is called using a pointer having type `D*`, `dp` is used to determine the access to the function `f()`. This call produces an error because `f()` is declared private in class D.

```
class B {  
public:  
    virtual void f();  
};  
  
class D : public B {  
private:  
    void f();  
};  
  
int main() {  
    D dobj;  
    B* bp = &dobj;  
    D* dp = &dobj;  
  
    // valid, virtual B::f() is public,  
    // D::f() is called  
    bp->f();  
  
    // error, D::f() is private  
    dp->f();  
}
```

Abstract Classes

C++ An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one *pure virtual function*. You declare a pure virtual function by using a *pure specifier* (= 0) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class::

```
class AB {
public:
    virtual void f() = 0;
};
```

Function `AB::f` is a pure virtual function. A function declaration cannot have both a pure specifier and a definition. For example, the compiler will not allow the following:

```
struct A {
    virtual void g() { } = 0;
};
```

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class. The following example demonstrates this:

```
struct A {
    virtual void f() = 0;
};

struct B : A {
    virtual void f() { }
};

// Error:
// Class A is an abstract class
// A g();

// Error:
// Class A is an abstract class
// void h(A);
A& i(A&);

int main() {

    // Error:
    // Class A is an abstract class
    // A a;

    A* pa;
    B b;

    // Error:
    // Class A is an abstract class
    // static_cast<A>(b);
}
```

Class `A` is an abstract class. The compiler would not allow the function declarations `A g()` or `void h(A)`, declaration of object `a`, nor the static cast of `b` to type `A`.

Abstract Classes

Virtual member functions are inherited. A class derived from an abstract base class will also be abstract unless you override each pure virtual function in the derived class.

For example:

```
class AB {
public:
    virtual void f() = 0;
};

class D2 : public AB {
    void g();
};

int main() {
    D2 d;
}
```


The compiler will not allow the declaration of object d because D2 is an abstract class; it inherited the pure virtual function f() from AB. The compiler will allow the declaration of object d if you define function D2::g().

Note that you can derive an abstract class from a non-abstract class, and you can override a non-pure virtual function with a pure virtual function.

You can call member functions from a constructor or destructor of an abstract class. However, the results of calling (directly or indirectly) a pure virtual function from its constructor are undefined. The following example demonstrates this:

```
struct A {
    A() {
        direct();
        indirect();
    }
    virtual void direct() = 0;
    virtual void indirect() { direct(); }
};
```

The default constructor of A calls the pure virtual function direct() both directly and indirectly (through indirect()).

 The VisualAge C++ compiler issues a warning for the direct call to the pure virtual function. The compiler does not issue a warning for the indirect call to the pure virtual function.

Chapter 15. Special Member Functions

C++ The default constructor, destructor, copy constructor, and copy assignment operator are *special member functions*. These functions create, destroy, convert, initialize, and copy class objects.

RELATED REFERENCES

- “Constructors” on page 342
- “Destructors” on page 350
- “Conversion by Constructor” on page 360
- “Conversion Functions” on page 361
- “Copy Constructors” on page 362

Constructors and Destructors Overview

C++ Because classes have complicated internal structures, including data and functions, object initialization and cleanup for classes is much more complicated than it is for simple data structures. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deallocation of memory for objects.

Like other member functions, constructors and destructors are declared within a class declaration. They can be defined inline or external to the class declaration. Constructors can have default arguments. Unlike other member functions, constructors can have member initialization lists. The following restrictions apply to constructors and destructors:

- Constructors and destructors do not have return types nor can they return values.
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
- Constructors cannot be declared with the keyword **virtual**.
- Constructors and destructors cannot be declared **static**, **const**, or **volatile**.
- Unions cannot contain class objects that have constructors or destructors.

Constructors and destructors obey the same access rules as member functions. For example, if you declare a constructor with protected access, only derived classes and friends can use it to create class objects.

The compiler automatically calls constructors when defining class objects and calls destructors when class objects go out of scope. A constructor does not allocate memory for the class object its **this** pointer refers to, but may allocate storage for more objects than its class object refers to. If memory allocation is required for objects, constructors can explicitly call the **new** operator. During cleanup, a destructor may release objects allocated by the corresponding constructor. To release objects, use the **delete** operator.

Derived classes do not inherit constructors or destructors from their base classes, but they do call the constructor and destructor of base classes. Destructors can be declared with the keyword **virtual**.

Constructors are also called when local or temporary class objects are created, and destructors are called when local or temporary objects go out of scope.

You can call member functions from constructors or destructors. You can call a virtual function, either directly or indirectly, from a constructor or destructor of a class A. In this case, the function called is the one defined in A or a base class of A, but not a function overridden in any class derived from A. This avoids the possibility of accessing an unconstructed object from a constructor or destructor. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "void A::f()" << endl; }
    virtual void g() { cout << "void A::g()" << endl; }
    virtual void h() { cout << "void A::h()" << endl; }
};

struct B : A {
    virtual void f() { cout << "void B::f()" << endl; }
    B() {
        f();
        g();
        h();
    }
};

struct C : B {
    virtual void f() { cout << "void C::f()" << endl; }
    virtual void g() { cout << "void C::g()" << endl; }
    virtual void h() { cout << "void C::h()" << endl; }
};

int main() {
    C obj;
}
```

The following is the output of the above example:

```
void B::f()
void A::g()
void A::h()
```

The constructor of B does not call any of the functions overridden in C because C has been derived from B, although the example creates an object of type C named obj.

You can use the **typeid** or the **dynamic_cast** operator in constructors or destructors, as well as member initializers of constructors.

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69
- “static Storage Class Specifier” on page 42
- “C++ new Operator” on page 119
- “C++ delete Operator” on page 122
- “Free Store” on page 353
- “The typeid Operator” on page 139
- “dynamic_cast Operator” on page 111

Constructors

C++ A *constructor* is a member function with the same name as its class. For example:



```
class X {
public:
    X();    // constructor for class X
};
```

Constructors are used to create, and can initialize, objects of their class type.

You cannot declare a constructor as **virtual** or **static**, nor can you declare a constructor as **const**, **volatile**, or **const volatile**.

You do not specify a return type for a constructor. A return statement in the body of a constructor cannot have a return value.

Default Constructors

 A *default constructor* is a constructor that either has no parameters, or if it has parameters, *all* the parameters have default values.

If no user-defined constructor exists for a class A and one is needed, the compiler implicitly *declares* a constructor `A::A()`. This constructor is an inline public member of its class. The compiler will implicitly *define* `A::A()` when the compiler uses this constructor to create an object of type A. The constructor will have no constructor initializer and a null body.

The compiler first implicitly defines the implicitly declared constructors of the base classes and nonstatic data members of a class A before defining the implicitly declared constructor of A. No default constructor is created for a class that has any constant or reference type members.

A constructor of a class A is *trivial* if all the following are true:

- It is implicitly defined
- A has no virtual functions and no virtual base classes
- All the direct base classes of A have trivial constructors
- The classes of all the nonstatic data members of A have trivial constructors

If any of the above are false, then the constructor is *nontrivial*.

A union member cannot be of a class type that has a nontrivial constructor.

Like all functions, a constructor can have default arguments. They are used to initialize member objects. If default values are supplied, the trailing arguments can be omitted in the expression list of the constructor. Note that if a constructor has any arguments that do not have default values, it is not a default constructor.

A *copy constructor* for a class A is a constructor whose first parameter is of type `A&`, `const A&`, `volatile A&`, or `const volatile A&`. Copy constructors are used to make a copy of one class object from another class object of the same class type. You cannot use a copy constructor with an argument of the same type as its class; you must use a reference. You can provide copy constructors with additional parameters as long as they all have default arguments. If a user-defined copy constructor does not exist for a class and one is needed, the compiler implicitly creates a copy constructor, with public access, for that class. A copy constructor is not created for a class if any of its members or base classes have an inaccessible copy constructor.

The following code fragment shows two classes with constructors, default constructors, and copy constructors:

```

class X {
public:

    // default constructor, no arguments
    X();

    // constructor
    X(int, int , int = 0);

    // copy constructor
    X(const X&);

    // error, incorrect argument type
    X(X);
};

class Y {
public:

    // default constructor with one
    // default argument
    Y( int = 0);

    // default argument
    // copy constructor
    Y(const Y&, int = 0);
};

```

RELATED REFERENCES

- “Copy Constructors” on page 362

Explicit Initialization with Constructors

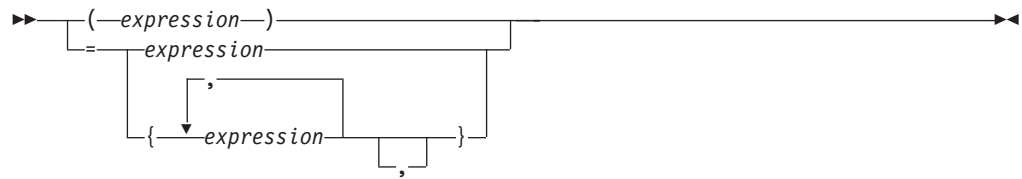
C++ A class object with a constructor must be explicitly initialized or have a default constructor. Except for aggregate initialization, explicit initialization using a constructor is the only way to initialize nonstatic constant and reference class members.

A class object that has no constructors, no virtual functions, no private or protected members, and no base classes is called an *aggregate*. Examples of aggregates are C-style structures and unions.

You explicitly initialize a class object when you create that object. There are two ways to initialize a class object:

- Using a parenthesized expression list. The compiler calls the constructor of the class using this list as the constructor’s argument list.
- Using a single initialization value and the = operator. Because this type of expression is an initialization, not an assignment, the assignment operator function, if one exists, is not called. The type of the single argument must match the type of the first argument to the constructor. If the constructor has remaining arguments, these arguments must have default values.
-

The syntax for an initializer that explicitly initializes a class object with a constructor is:



The following example shows the declaration and use of several constructors that explicitly initialize class objects:

CCNX13A

```
// This example illustrates explicit initialization
// by constructor.
#include <iostream>
using namespace std;

class complx {
    double re, im;
public:

    // default constructor
    complx() : re(0), im(0) { }

    // copy constructor
    complx(const complx& c) { re = c.re; im = c.im; }

    // constructor with default trailing argument
    complx( double r, double i = 0.0) { re = r; im = i; }

    void display() {
        cout << "re = " << re << " im = " << im << endl;
    }
};

int main() {

    // initialize with complx(double, double)
    complx one(1);

    // initialize with a copy of one
    // using complx::complx(const complx&)
    complx two = one;

    // construct complx(3,4)
    // directly into three
    complx three = complx(3,4);

    // initialize with default constructor
    complx four;

    // complx(double, double) and construct
    // directly into five
    complx five = 5;

    one.display();
    two.display();
    three.display();
    four.display();
    five.display();
}
```

The above example produces the following output:

```

re = 1 im = 0
re = 1 im = 0
re = 3 im = 4
re = 0 im = 0
re = 5 im = 0

```

RELATED REFERENCES

- “Structures” on page 51
- “Unions” on page 59

Initializing Base Classes and Members

C++ Constructors can initialize their members in two different ways. A constructor can use the arguments passed to it to initialize member variables in the constructor definition:

```

complex(double r, double i = 0.0) { re = r; im = i; }

```

Or a constructor can have an *initializer list* within the definition but prior to the function body:

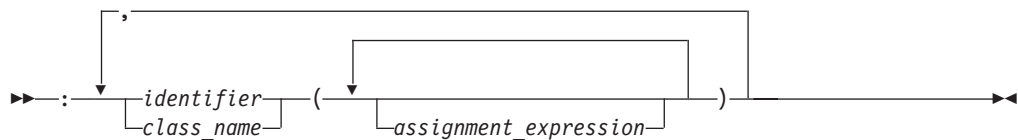
```

complex(double r, double i = 0) : re(r), im(i) { /* ... */ }

```

Both methods assign the argument values to the appropriate data members of the class.

The syntax for a constructor initializer list is:



Include the initialization list as part of the function definition, not as part of the constructor declaration. For example:

```

#include <iostream>
using namespace std;

class B1 {
    int b;
public:
    B1() { cout << "B1::B1()" << endl; };

    // inline constructor
    B1(int i) : b(i) { cout << "B1::B1(int)" << endl; }
};

class B2 {
    int b;
protected:
    B2() { cout << "B1::B1()" << endl; }

    // noninline constructor
    B2(int i);
};

// B2 constructor definition including initialization list
B2::B2(int i) : b(i) { cout << "B2::B2(int)" << endl; }

class D : public B1, public B2 {

```

```

    int d1, d2;
public:
    D(int i, int j) : B1(i+1), B2(), d1(i) {
        cout << "D1::D1(int, int)" << endl;
        d2 = j;}
};

int main() {
    D obj(1, 2);
}

```

The following is the output of the above example:

```

B1::B1(int)
B1::B1()
D1::D1(int, int)

```

If you do not explicitly initialize a base class or member that has constructors by calling a constructor, the compiler automatically initializes the base class or member with a default constructor. In the above example, if you leave out the call `B2()` in the constructor of class `D` (as shown below), a constructor initializer with an empty expression list is automatically created to initialize `B2`. The constructors for class `D`, shown above and below, result in the same construction of an object of class `D`:

```

class D : public B1, public B2 {
    int d1, d2;
public:

    // call B2() generated by compiler
    D(int i, int j) : B1(i+1), d1(i) {
        cout << "D1::D1(int, int)" << endl;
        d2 = j;}
};

```

In the above example, the compiler will automatically call the default constructor for `B2()`.

Note that you must declare constructors as public or protected to enable a derived class to call them. For example:

```

class B {
    B() { }
};

class D : public B {

    // error: implicit call to private B() not allowed
    D() { }
};

```

The compiler would not allow the definition of `D::D()` because this constructor cannot access the private constructor `B::B()`.

You must initialize the following with an initializer list: base classes with no default constructors, reference data members, non-static const data members, or a class type which contains a constant data member. The following example demonstrates this:

```

class A {
public:
    A(int) { }
};

class B : public A {
    static const int i;
}

```

```

    const int j;
    int &k;
public:
    B(int& arg) : A(0), j(1), k(arg) { }
};

int main() {
    int x = 0;
    B obj(x);
};

```

The data members `j` and `k`, as well as the base class `A` must be initialized in the initializer list of the constructor of `B`.

You can use data members when initializing members of a class. The following example demonstrate this:

```

struct A {
    int k;
    A(int i) : k(i) { }
};
struct B: A {
    int x;
    int i;
    int j;
    int& r;
    B(int i): r(x), A(i), j(this->i), i(i) { }
};

```

The constructor `B(int i)` initializes the following:

- `B::x` to refer to `A::x`
- Class `A` with the value of the argument to `B(int i)`
- `B::j` with the value of `B::i`
- `B::i` with the value of the argument to `B(int i)`

You can also call member functions (including virtual member functions) or use the operators **typeid** or **dynamic_cast** when initializing members of a class. However if you perform any of these operations in a member initialization list before all base classes have been initialized, the behavior is undefined. The following example demonstrates this:

```

#include <iostream>
using namespace std;

struct A {
    int i;
    A(int arg) : i(arg) {
        cout << "Value of i: " << i << endl;
    }
};

struct B : A {
    int j;
    int f() { return i; }
    B();
};

B::B() : A(f()), j(1234) {
    cout << "Value of j: " << j << endl;
}

int main() {
    B obj;
}

```

The output of the above example would be similar to the following:

```
Value of i: 8  
Value of j: 1234
```

The behavior of the initializer `A(f())` in the constructor of `B` is undefined. The run time will call `B::f()` and try to access `A::i` even though the base `A` has not been initialized.

The following example is the same as the previous example except that the initializers of `B::B()` have different arguments:

```
#include <iostream>  
using namespace std;  
  
struct A {  
    int i;  
    A(int arg) : i(arg) {  
        cout << "Value of i: " << i << endl;  
    }  
};  
  
struct B : A {  
    int j;  
    int f() { return i; }  
    B();  
};  
  
B::B() : A(5678), j(f()) {  
    cout << "Value of j: " << j << endl;  
}  
  
int main() {  
    B obj;  
}
```

The following is the output of the above example:

```
Value of i: 5678  
Value of j: 5678
```

The behavior of the initializer `j(f())` in the constructor of `B` is well-defined. The base class `A` is already initialized when `B::j` is initialized.

RELATED REFERENCES

- “Default Constructors” on page 343
- “The typeid Operator” on page 139
- “dynamic_cast Operator” on page 111

Construction Order of Derived Class Objects

> C++ When a derived class object is created using constructors, it is created in the following order:

1. Virtual base classes are initialized, in the order they appear in the base list.
2. Nonvirtual base classes are initialized, in declaration order.
3. Class members are initialized in declaration order (regardless of their order in the initialization list).
4. The body of the constructor is executed.

The following example demonstrates this:

```

#include <iostream>
using namespace std;
struct V {
    V() { cout << "V()" << endl; }
};
struct V2 {
    V2() { cout << "V2()" << endl; }
};
struct A {
    A() { cout << "A()" << endl; }
};
struct B : virtual V {
    B() { cout << "B()" << endl; }
};
struct C : B, virtual V2 {
    C() { cout << "C()" << endl; }
};
struct D : C, virtual V {
    A obj_A;
    D() { cout << "D()" << endl; }
};
int main() {
    D c;
}

```

The following is the output of the above example:

```

V()
V2()
B()
C()
A()
D()

```

The above output lists the order in which the C++ run time calls the constructors to create an object of type D.

RELATED REFERENCES

- “Virtual Base Classes” on page 328

Destructors

C++ *Destructors* are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

A destructor is a member function with the same name as its class prefixed by a `~` (tilde). For example:

```

class X {
public:
    // Constructor for class X
    X();
    // Destructor for class X
    ~X();
};

```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared **const**, **volatile**, **const volatile** or **static**. A destructor can be declared **virtual** or pure **virtual**.

If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared constructor is an inline public member of its class.

The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class A has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for A:

```
~A::A() { }
```

The compiler first implicitly defines the implicitly declared destructors of the base classes and nonstatic data members of a class A before defining the implicitly declared destructor of A

A destructor of a class A is *trivial* if all the following are true:

- It is implicitly defined
- All the direct base classes of A have trivial destructors
- The classes of all the nonstatic data members of A have trivial destructors

If any of the above are false, then the destructor is *nontrivial*.

A union member cannot be of a class type that has a nontrivial destructor.

Class members that are class types can have their own destructors. Both base and derived classes can have destructors, although destructors are not inherited. If a base class A or a member of A has a destructor, and a class derived from A does not declare a destructor, a default destructor is generated.

The default destructor calls the destructors of the base class and members of the derived class.

The destructors of base classes and members are called in the reverse order of the completion of their constructor:

1. The destructor for a class object is called before destructors for members and bases are called.
2. Destructors for nonstatic members are called before destructors for base classes are called.
3. Destructors for nonvirtual base classes are called before destructors for virtual base classes are called.

When an exception is thrown for a class object with a destructor, the destructor for the temporary object thrown is not called until control passes out of the catch block.

Destructors are implicitly called when an automatic object (a local object that has been declared **auto** or **register**, or not declared as **static** or **extern**) or temporary object passes out of scope. They are implicitly called at program termination for constructed external and static objects. Destructors are invoked when you use the **delete** operator for objects created with the **new** operator.

For example:

```
#include <string>

class Y {
private:
    char * string;
    int number;
```

```

public:
    // Constructor
    Y(const char*, int);
    // Destructor
    ~Y() { delete[] string; }
};

// Define class Y constructor
Y::Y(const char* n, int a) {
    string = strcpy(new char[strlen(n) + 1 ], n);
    number = a;
}

int main () {
    // Create and initialize
    // object of class Y
    Y yobj = Y("somestring", 10);

    // ...

    // Destructor ~Y is called before
    // control returns from main()
}

```

You can use a destructor explicitly to destroy objects, although this practice is not recommended. However to destroy an object created with the placement **new** operator, you can explicitly call the object's destructor. The following example demonstrates this:

```

#include <new>
#include <iostream>
using namespace std;
class A {
public:
    A() { cout << "A::A()" << endl; }
    ~A() { cout << "A::~ A()" << endl; }
};

int main () {
    char* p = new char[sizeof(A)];
    A* ap = new (p) A;
    ap->A:: A();
    delete [] p;
}

```

The statement `A* ap = new (p) A` dynamically creates a new object of type `A` not in the free store but in the memory allocated by `p`. The statement `delete [] p` will delete the storage allocated by `p`, but the run time will still believe that the object pointed to by `ap` still exists until you explicitly call the destructor of `A` (with the statement `ap->A::~ A()`).

Nonclass types have a *pseudo destructor*. The following example calls the pseudo destructor for an integer type:

```

typedef int I;
int main() {
    I x = 10;
    x.I::~ I();
    x = 20;
}

```

The call to the pseudo destructor, `x.I::~ I()`, has no effect at all. Object `x` has not been destroyed; the assignment `x = 20` is still valid. Because pseudo destructors

require the syntax for explicitly calling a destructor for a nonclass type to be valid, you can write code without having to know whether or not a destructor exists for a given type.

RELATED REFERENCES

- “C++ new Operator” on page 119
- “C++ delete Operator” on page 122
- “Temporary Objects” on page 357

Free Store

C++ *Free store* is a pool of memory available for you to allocate (and deallocate) storage for objects during the execution of your program. The **new** and **delete** operators are used to allocate and deallocate free store, respectively.

You can define your own versions of **new** and **delete** for a class by overloading them. You can declare the **new** and **delete** operators with additional parameters. When **new** and **delete** operate on class objects, the class member operator functions **new** and **delete** are called, if they have been declared.

If you create a class object with the **new** operator, one of the operator functions **operator new()** or **operator new[]()** (if they have been declared) is called to create the object. An **operator new()** or **operator new[]()** for a class is always a static class member, even if it is not declared with the keyword **static**. It has a return type **void*** and its first parameter must be the size of the object type and have type **std::size_t**. It cannot be virtual.

Type **std::size_t** is an implementation-dependent unsigned integral type defined in the standard library header `<cstddef>`.

When you overload the **new** operator, you must declare it as a class member, returning type **void***, with its first parameter of type **std::size_t**, as described above. You can declare additional parameters in the declaration of **operator new()** or **operator new[]()**. Use the placement syntax to specify values for these parameters in an allocation expression.

The following example overloads two operator **new** functions:

- `X::operator new(size_t sz)`: This overloads the default **new** operator by allocating memory with the C function **malloc()**, and throwing a string (instead of **std::bad_alloc**) if **malloc()** fails.
- `X::operator new(size_t sz, int location)`: This function takes an additional integer parameter, `location`. This function implements a very simplistic “memory manager” that manages the storage of up to three X objects.

Static array `X::buffer` holds three Node objects. Each Node object contains a pointer to an X object named `data` and a boolean variable named `filled`. Each X object stores an integer called `number`.

When you use this **new** operator, you pass the argument `location` which indicates the array location of `buffer` where you want to “create” your new X object. If the array location is not “filled” (the `data` member of `filled` is equal to **false** at that array location), the **new** operator returns a pointer pointing to the X object located at `buffer[location]`.

```
#include <new>
#include <iostream>
```

```

using namespace std;

class X;

struct Node {
    X* data;
    bool filled;
    Node() : filled(false) { }
};

class X {
    static Node buffer[];

public:

    int number;

    enum {size = 3};

    void* operator new(size_t sz) throw (const char*) {
        void* p = malloc(sz);
        if (sz == 0) throw "Error: malloc() failed";
        cout << "X::operator new(size_t)" << endl;
        return p;
    }

    void *operator new(size_t sz, int location) throw (const char*) {
        cout << "X::operator new(size_t, " << location << ")" << endl;
        void* p = 0;
        if (location < 0 || location >= size || buffer[location].filled == true) {
            throw "Error: buffer location occupied";
        }
        else {
            p = malloc(sizeof(X));
            if (p == 0) throw "Error: Creating X object failed";
            buffer[location].filled = true;
            buffer[location].data = (X*) p;
        }
        return p;
    }

    static void printbuffer() {
        for (int i = 0; i < size; i++) {
            cout << buffer[i].data->number << endl;
        }
    }
};

Node X::buffer[size];

int main() {
    try {
        X* ptr1 = new X;
        X* ptr2 = new(0) X;
        X* ptr3 = new(1) X;
        X* ptr4 = new(2) X;
        ptr2->number = 10000;
        ptr3->number = 10001;
        ptr4->number = 10002;
        X::printbuffer();
        X* ptr5 = new(0) X;
    }
    catch (const char* message) {
        cout << message << endl;
    }
}

```

The following is the output of the above example:

```
X::operator new(size_t)
X::operator new(size_t, 0)
X::operator new(size_t, 1)
X::operator new(size_t, 2)
10000
10001
10002
X::operator new(size_t, 0)
Error: buffer location occupied
```

The statement `X* ptr1 = new X` calls `X::operator new(sizeof(X))`. The statement `X* ptr2 = new(0) X` calls `X::operator new(sizeof(X),0)`.

The **delete** operator destroys an object created by the **new** operator. The operand of **delete** must be a pointer returned by **new**. If **delete** is called for an object with a destructor, the destructor is invoked before the object is deallocated.

If you destroy a class object with the **delete** operator, the operator function **operator delete()** or **operator delete[]()** (if they have been declared) is called to destroy the object. An **operator delete()** or **operator delete[]()** for a class is always a static member, even if it is not declared with the keyword **static**. Its first parameter must have type **void***. Because **operator delete()** and **operator delete[]()** have a return type **void**, they cannot return a value.

The following example shows the declaration and use of the operator functions **operator new()** and **operator delete()**:

```
#include <cstdlib>
#include <iostream>
using namespace std;

class X {
public:
    void* operator new(size_t sz) throw (const char*) {
        void* p = malloc(sz);
        if (p == 0) throw "malloc() failed";
        return p;
    }

    // single argument
    void operator delete(void* p) {
        cout << "X::operator delete(void*)" << endl;
        free(p);
    }
};

class Y {
    int filler[100];
public:

    // two arguments
    void operator delete(void* p, size_t sz) throw (const char*) {
        cout << "Freeing " << sz << " byte(s)" << endl;
        free(p);
    }
};

int main() {
    X* ptr = new X;

    // call X::operator delete(void*)
```

```

delete ptr;

Y* yptr = new Y;

// call Y::operator delete(void*, size_t)
// with size of Y as second argument
delete yptr;
}

```

The above example will generate output similar to the following:

```

X::operator delete(void*)
Freeing 400 byte(s)

```

The statement `delete ptr` calls `X::operator delete(void*)`. The statement `delete yptr` calls `Y::operator delete(void*, size_t)`.

The result of trying to access a deleted object is undefined because the value of the object can change after deletion.

If **new** and **delete** are called for a class object that does not declare the operator functions **new** and **delete**, or they are called for a nonclass object, the global operators **new** and **delete** are used. The global operators **new** and **delete** are provided in the C++ library.

The C++ operators for allocating and deallocating arrays of class objects are **operator new[]()** and **operator delete[]()**.

You cannot declare the **delete** operator as virtual. However you can add polymorphic behavior to your **delete** operators by declaring the destructor of a base class as virtual. The following example demonstrates this:

```

#include <iostream>
using namespace std;

struct A {
    virtual ~A() { cout << "~A()" << endl; };
    void operator delete(void* p) {
        cout << "A::operator delete" << endl;
        free(p);
    }
};

struct B : A {
    void operator delete(void* p) {
        cout << "B::operator delete" << endl;
        free(p);
    }
};

int main() {
    A* ap = new B;
    delete ap;
}

```

The following is the output of the above example:

```

~A()
B::operator delete

```

The statement `delete ap` uses the **delete** operator from class B instead of class A because the destructor of A has been declared as virtual.

Although you can get polymorphic behavior from the **delete** operator, the **delete** operator that is statically visible must still be accessible even though another **delete** operator might be called. For example, in the above example, the function `A::operator delete(void*)` must be accessible even though the example calls `B::operator delete(void*)` instead.

Virtual destructors do not have any affect on deallocation operators for arrays (**operator delete[]()**). The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
    virtual ~A() { cout << "~A()" << endl; }
    void operator delete[](void* p, size_t) {
        cout << "A::operator delete[]" << endl;
        ::delete [] p;
    }
};

struct B : A {
    void operator delete[](void* p, size_t) {
        cout << "B::operator delete[]" << endl;
        ::delete [] p;
    }
};

int main() {
    A* bp = new B[3];
    delete[] bp;
};
```


The behavior of the statement `delete[] bp` is undefined.

When you overload the **delete** operator, you must declare it as class member, returning type **void**, with the first parameter having type **void***, as described above. You can add a second parameter of type **size_t** to the declaration. You can only have one **operator delete()** or **operator delete[]()** for a single class.

RELATED REFERENCES

- “C++ new Operator” on page 119
- “C++ delete Operator” on page 122
- “Allocation and Deallocation Functions” on page 123

Temporary Objects

 It is sometimes necessary for the compiler to create temporary objects. They are used during reference initialization and during evaluation of expressions including standard type conversions, argument passing, function returns, and evaluation of the **throw** expression.

When a temporary object is created to initialize a reference variable, the name of the temporary object has the same scope as that of the reference variable. When a temporary object is created during the evaluation of a full-expression (an expression that is not a subexpression of another expression), it is destroyed as the last step in its evaluation that lexically contains the point where it was created.

There are two exceptions in the destruction of full-expressions:

- The expression appears as an initializer for a declaration defining an object: the temporary object is destroyed when the initialization is complete.
- A reference is bound to a temporary object: the temporary object is destroyed at the end of the reference's lifetime.

If a temporary object is created for a class with constructors, the compiler calls the appropriate (matching) constructor to create the temporary object.

When a temporary object is destroyed and a destructor exists, the compiler calls the destructor to destroy the temporary object. When you exit from the scope in which the temporary object was created, it is destroyed. If a reference is bound to a temporary object, the temporary object is destroyed when the reference passes out of scope unless it is destroyed earlier by a break in the flow of control. For example, a temporary object created by a constructor initializer for a reference member is destroyed on leaving the constructor.

The ISO C++ definition permits an implementation that eliminates the construction of such temporary objects in cases in which they are redundant.

► VAC++ The VisualAge C++ compiler takes advantage of this fact to create more efficient optimized code. Take this into consideration when debugging your programs, especially for memory problems.

RELATED REFERENCES

- “Arguments of catch Blocks” on page 407
- “Initializing References” on page 93
- “Cast Expressions” on page 135
- “Function Return Values” on page 171

User-Defined Conversions

► C++ *User-defined conversions* allow you to specify object conversions with constructors or with conversion functions. User-defined conversions are implicitly used in addition to standard conversions for conversion of initializers, functions arguments, function return values, expression operands, expressions controlling iteration, selection statements, and explicit type conversions.

There are two types of user-defined conversions:

- Conversion by constructor
- Conversion functions

The compiler can use only one user-defined conversion (either a conversion constructor or a conversion function) when implicitly converting a single value. The following example demonstrates this:

```
class A {
    int x;
public:
    operator int() { return x; };
};

class B {
    A y;
public:
    operator A() { return y; };
};
```



```

int main () {
    B b_obj;
    // int i = b_obj;
    int j = A(b_obj);
}

```

The compiler would not allow the statement `int i = b_obj`. The compiler would have to implicitly convert `b_obj` into an object of type `A` (with `B::operator A()`), then implicitly convert that object to an integer (with `A::operator int()`). The statement `int j = A(b_obj)` explicitly converts `b_obj` into an object of type `A`, then implicitly converts that object to an integer.

User-defined conversions must be unambiguous, or they are not called. A conversion function in a derived class does not hide another conversion function in a base class unless both conversion functions convert to the same type. Function overload resolution selects the most appropriate conversion function. The following example demonstrates this:

```

class A {
    int a_int;
    char* a_carp;
public:
    operator int() { return a_int; }
    operator char*() { return a_carp; }
};

class B : public A {
    float b_float;
    char* b_carp;
public:
    operator float() { return b_float; }
    operator char*() { return b_carp; }
};

int main () {
    B b_obj;
    // long a = b_obj;
    char* c_p = b_obj;
}

```

The compiler would not allow the statement `long a = b_obj`. The compiler could either use `A::operator int()` or `B::operator float()` to convert `b_obj` into a **long**. The statement `char* c_p = b_obj` uses `B::operator char*()` to convert `b_obj` into a **char*** because `B::operator char*()` hides `A::operator char*()`.

When you call a constructor with an argument and you have not defined a constructor accepting that argument type, only standard conversions are used to convert the argument to another argument type acceptable to a constructor for that class. No other constructors or conversions functions are called to convert the argument to a type acceptable to a constructor defined for that class. The following example demonstrates this:

```

class A {
public:
    A() { }
    A(int) { }
};

int main() {
    A a1 = 1.234;
    // A moocow = "text string";
}

```

The compiler allows the statement `A a1 = 1.234`. The compiler uses the standard conversion of converting 1.234 into an **int**, then implicitly calls the converting constructor `A(int)`. The compiler would not allow the statement `A moocow = "text string"`; converting a text string to an integer is not a standard conversion.

RELATED REFERENCES

- “Standard Type Conversions” on page 144

Conversion by Constructor

C++ A *converting constructor* is a constructor that can be called with one parameter, but is not declared without the function specifier **explicit**. The compiler uses converting constructors to convert objects from the type of the first parameter to the type of the converting constructor's class. The following example demonstrates this:

```
class Y {
    int a, b;
    char* name;
public:
    Y(int i) { };
    Y(const char* n, int j = 0) { };
};

void add(Y) { };

int main() {

    // equivalent to
    // obj1 = Y(2)
    Y obj1 = 2;

    // equivalent to
    // obj2 = Y("somestring",0)
    Y obj2 = "somestring";

    // equivalent to
    // obj1 = Y(10)
    obj1 = 10;

    // equivalent to
    // add(Y(5))
    add(5);
}
```

The above example has the following two converting constructors:

- `Y(int i)` which is used to convert integers to objects of class `Y`.
- `Y(const char* n, int j = 0)` which is used to convert pointers to strings to objects of class `Y`.

The compiler will not implicitly convert types as demonstrated above with constructors declared with the **explicit** keyword. The compiler will only use explicitly declared constructors in **new** expressions, the **static_cast** expressions and explicit casts, and the initialization of bases and members. The following example demonstrates this:

```
class A {
public:
    explicit A() { };
    explicit A(int) { };
};
```

```

int main() {
    A z;
    // A y = 1;
    A x = A(1);
    A w(1);
    A* v = new A(1);
    A u = (A)1;
    A t = static_cast<A>(1);
}

```

The compiler would not allow the statement `A y = 1` because this is an implicit conversion; class `A` has no conversion constructors.

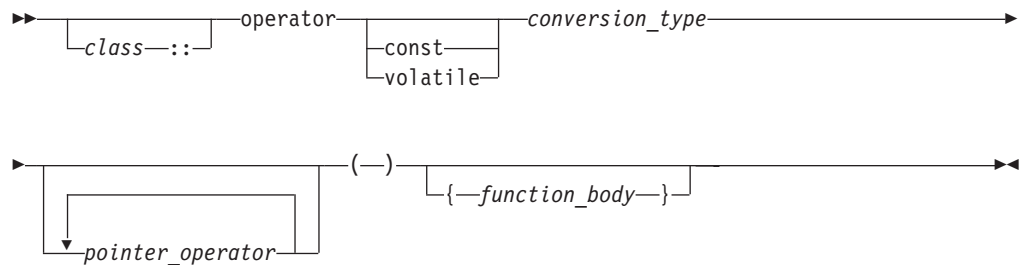
A copy constructor is a converting constructor.

RELATED REFERENCES

- “The explicit Keyword” on page 150
- “C++ new Operator” on page 119
- “static_cast Operator” on page 108
- “Cast Expressions” on page 135

Conversion Functions

C++ You can define a member function of a class, called a *conversion function*, that converts from the type of its class to another specified type.



A conversion function that belongs to a class `X` specifies a conversion from the class type `X` to the type specified by the *conversion_type*. The following code fragment shows a conversion function called `operator int()`:

```

class Y {
    int b;
public:
    operator int();
};
Y::operator int() {
    return b;
}
void f(Y obj) {
    int i = int(obj);
    int j = (int)obj;
    int k = i + obj;
}

```

All three statements in function `f(Y)` use the conversion function `Y::operator int()`.

Classes, enumerations, **typedef** names, function types, or array types cannot be declared or defined in the *conversion_type*. You cannot use a conversion function to convert an object of type A to type A, a base class of A, or void.

Conversion functions have no arguments, and the return type is implicitly the conversion type. Conversion functions can be inherited. You can have virtual conversion functions but not static ones.

Copy Constructors

C++ The *copy constructor* lets you create a new object from an existing one by initialization. A copy constructor of a class A is a nontemplate constructor in which the first parameter is of type A&, const A&, volatile A&, or const volatile A&, and the rest of its parameters (if there are any) have default values.

If you do not declare a copy constructor for a class A, the compiler will implicitly declare one for you, which will be an inline public member.

The following example demonstrates implicitly defined and user-defined copy constructors:

```
#include <iostream>
using namespace std;

struct A {
    int i;
    A() : i(10) { }
};

struct B {
    int j;
    B() : j(20) {
        cout << "Constructor B(), j = " << j << endl;
    }

    B(B& arg) : j(arg.j) {
        cout << "Copy constructor B(B&), j = " << j << endl;
    }

    B(const B&, int val = 30) : j(val) {
        cout << "Copy constructor B(const B&, int), j = " << j << endl;
    }
};

struct C {
    C() { }
    C(C&) { }
};

int main() {
    A a;
    A a1(a);
    B b;
    const B b_const;
    B b1(b);
    B b2(b_const);
    const C c_const;
    // C c1(c_const);
}
```

The following is the output of the above example:

```
Constructor B(), j = 20  
Constructor B(), j = 20  
Copy constructor B(B&), j = 20  
Copy constructor B(const B&, int), j = 30
```

The statement `A a1(a)` creates a new object from `a` with an implicitly defined copy constructor. The statement `B b1(b)` creates a new object from `b` with the user-defined copy constructor `B::B(B&)`. The statement `B b2(b_const)` creates a new object with the copy constructor `B::B(const B&, int)`. The compiler would not allow the statement `C c1(c_const)` because a copy constructor that takes as its first parameter an object of type `const C&` has not been defined.

The implicitly declared copy constructor of a class `A` will have the form `A::A(const A&)` if the following are true:

- The direct and virtual bases of `A` have copy constructors whose first parameters have been qualified with **const** or **const volatile**
- The nonstatic class type or array of class type data members of `A` have copy constructors whose first parameters have been qualified with **const** or **const volatile**

If the above are not true for a class `A`, the compiler will implicitly declare a copy constructor with the form `A::A(A&)`.

The compiler cannot allow a program in which the compiler must implicitly define a copy constructor for a class `A` and one or more of the following are true:

- Class `A` has a nonstatic data member of a type which has an inaccessible or ambiguous copy constructor.
- Class `A` is derived from a class which has an inaccessible or ambiguous copy constructor.

The compiler will implicitly define an implicitly declared constructor of a class `A` if you initialize an object of type `A` or an object derived from class `A`.

An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

RELATED REFERENCES

- “Constructors and Destructors Overview” on page 341

Copy Assignment Operators

► **C++** The *copy assignment operator* lets you create a new object from an existing one by initialization. A copy assignment operator of a class `A` is a nonstatic nontemplate member function that has one of the following forms:

- `A::operator=(A)`
- `A::operator=(A&)`
- `A::operator=(const A&)`
- `A::operator=(volatile A&)`
- `A::operator=(const volatile A&)`

If you do not declare a copy assignment operator for a class `A`, the compiler will implicitly declare one for you which will be inline public.

The following example demonstrates implicitly defined and user-defined copy assignment operators:

```
#include <iostream>
using namespace std;

struct A {
    A& operator=(const A&) {
        cout << "A::operator=(const A&)" << endl;
        return *this;
    }

    A& operator=(A&) {
        cout << "A::operator=(A&)" << endl;
        return *this;
    }
};

class B {
    A a;
};

struct C {
    C& operator=(C&) {
        cout << "C::operator=(C&)" << endl;
        return *this;
    }
    C() { }
};

int main() {
    B x, y;
    x = y;

    A w, z;
    w = z;

    C i;
    const C j();
    // i = j;
}
```

The following is the output of the above example:

```
A::operator=(const A&)
A::operator=(A&)
```

The assignment `x = y` calls the implicitly defined copy assignment operator of `B`, which calls the user-defined copy assignment operator `A::operator=(const A&)`. The assignment `w = z` calls the user-defined operator `A::operator=(A&)`. The compiler will not allow the assignment `i = j` because an operator `C::operator=(const C&)` has not been defined.

The implicitly declared copy assignment operator of a class `A` will have the form `A& A::operator=(const A&)` if the following are true:

- A direct or virtual base `B` of class `A` has a copy assignment operator whose parameter is of type `const B&`, `const volatile B&`, or `B`.
- A non-static class type data member of type `X` that belongs to class `A` has a copy constructor whose parameter is of type `const X&`, `const volatile X&`, or `X`.

If the above are not true for a class `A`, the compiler will implicitly declare a copy assignment operator with the form `A& A::operator=(A&)`.

The implicitly declared copy assignment operator returns a reference to the operator's argument.

The copy assignment operator of a derived class hides the copy assignment operator of its base class.

The compiler cannot allow a program in which the compiler must implicitly define a copy assignment operator for a class A and one or more of the following are true:

- Class A has a nonstatic data member of a **const** type or a reference type
- Class A has a nonstatic data member of a type which has an inaccessible copy assignment operator
- Class A is derived from a base class with an inaccessible copy assignment operator.

An implicitly defined copy assignment operator of a class A will first assign the direct base classes of A in the order that they appear in the definition of A. Next, the implicitly defined copy assignment operator will assign the nonstatic data members of A in the order of their declaration in the definition of A.

RELATED REFERENCES

- “Constructors and Destructors Overview” on page 341

Chapter 16. Templates

C++ A *template* describes a set of related classes or set of related functions in which a list of parameters in the declaration describe how the members of the set vary. The compiler generates new classes or functions when you supply arguments for these parameters; this process is called *template instantiation*. This class or function definition generated from a template and a set of template parameters is called a *specialization*.

Syntax — Template Declaration

► `template` \leftarrow `template_parameter_list` \rightarrow `declaration` \leftarrow `export` \leftarrow

The compiler accepts and silently ignores the `export` keyword on a template.

The *template_parameter_list* is a comma-separated list of the following kinds of template parameters:

- non-type
- type
- template

The *declaration* is one of the following::

- a declaration or definition of a function or a class
- a definition of a member function or a member class
- a definition of a static data member of a class template
- a definition of a static data member of a class nested within a class template
- a definition of a member template of a class or class template

The *identifier* of a *type* is defined to be a *type_name* in the scope of the template declaration. A template declaration can appear as a namespace scope or class scope declaration.

The following example demonstrates the use of a class template:

```
template<class L> class Key
{
    L k;
    L* kptr;
    int length;
public:
    Key(L);
    // ...
};
```

Suppose the following declarations appear later:

```
Key<int> i;
Key<char*> c;
Key<mytype> m;
```

The compiler would create three objects. The following table shows the definitions of these three objects if they were written out in source form as regular classes, not as templates:

<code>class Key<int> i;</code>	<code>class Key<char*> c;</code>	<code>class Key<mytype> m;</code>
<pre>class Key { int k; int * kptr; int length; public: Key(int); // ... };</pre>	<pre>class Key { char* k; char** kptr; int length; public: Key(char*); // ... };</pre>	<pre>class Key { mytype k; mytype* kptr; int length; public: Key(mytype); // ... };</pre>

Note that these three classes have different names. The arguments contained within the angle braces are not just the arguments to the class names, but part of the class names themselves. `Key<int>` and `Key<char*>` are class names.

RELATED REFERENCES

- “Template Instantiation” on page 387
- “Template Specialization” on page 390
- “Template Parameters”

Template Parameters

➤ **C++** There are three kinds of template parameters:

- type
- non-type
- template

You may interchange the keywords **class** and **typename** in a template parameter declaration. You cannot use storage class specifiers (**static** and **auto**) in a template parameter declaration.

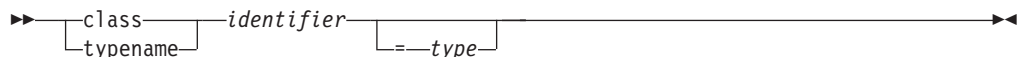
RELATED REFERENCES

- “Type Template Parameters”
- “Non-Type Template Parameters” on page 369
- “Template Template Parameters” on page 369

Type Template Parameters

➤ **C++** The following is the syntax of a type template parameter declaration:

Syntax —Type Template Parameter Declaration



The *identifier* is the name of a type.

RELATED REFERENCES

- “The Keyword `typename`” on page 398

Non-Type Template Parameters

C++ The syntax of a non-type template parameter is the same as a declaration of one of the following types:

- integral or enumeration
- pointer to object or pointer to function
- reference to object or reference to function
- pointer to member

Non-type template parameters that are declared as arrays or functions are converted to pointers or pointer to functions, respectively. The following example demonstrates this:

```
template<int a[4]> struct A { };
template<int f(int)> struct B { };
```

```
int q;
int g(int) {return 0;}
```

```
A<&q> x;
B<&g> y;
```

The type of `&q` is `int *`, and the type of `&g` is `int (*)(int)`.

You may qualify a non-type template parameter with **const** or **volatile**.

You cannot declare a non-type template parameter as a floating point, class, or void type.

Non-type template parameters are not lvalues.

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69
- “Lvalues and Rvalues” on page 99

Template Template Parameters

C++ The following is the syntax of a template template parameter declaration:

Syntax —Template Template Parameter Declaration

```
▶▶—template—<—template-parameter-list—>—class—                    —▶◀
                                          identifier      id-expression
```

The following example demonstrates a declaration and use of a template template parameter:

```
template<template <class T> class X> class A { };
template<class T> class B { };
```

```
A<B> a;
```

RELATED REFERENCES

- “Template Parameters” on page 368

Default Arguments for Template Parameters

C++ Template parameters may have default arguments. The set of default template arguments accumulates over all declarations of a given template. The following example demonstrates this:

```
template<class T, class U = int> class A;
template<class T = float, class U> class A;

template<class T, class U> class A {
    public:
        T x;
        U y;
};

A<> a;
```

The type of member `a.x` is **float**, and the type of `a.y` is **int**.

You cannot give default arguments to the same template parameters in different declarations in the same scope. For example, the compiler will not allow the following:

```
template<class T = char> class X;
template<class T = char> class X { };
```

If one template parameter has a default argument, then all template parameters following it must also have default arguments. For example, the compiler will not allow the following:

```
template<class T = char, class U, class V = int> class X { };
```

Template parameter `U` needs a default argument or the default for `T` must be removed.

The scope of a template parameter starts from the point of its declaration to the end of its template definition. This implies that you may use the name of a template parameter in other template parameter declarations and their default arguments. The following example demonstrates this:

```
template<class T = int> class A;
template<class T = float> class B;
template<class V, V obj> class A;
// a template parameter (T) used as the default argument
// to another template parameter (U)
template<class T, class U = T> class C { };
```

RELATED REFERENCES

- “Template Parameters” on page 368

Template Arguments

C++ There are three kinds of template arguments corresponding to the three types of template parameters:

- type
- non-type
- template

A template argument must match the type and form specified by the corresponding parameter declared in the template.

To use the default value of a template parameter, you omit the corresponding template argument. However, even if all template parameters have defaults, you still must use the `<>` brackets. For example, the following will yield a syntax error:

```
template<class T = int> class X { };
X<> a;
X b;
```

The last declaration, `X b`, will yield an error.

RELATED REFERENCES

- “Template Type Arguments”
- “Template Non-Type Arguments”
- “Template Template Arguments” on page 373

Template Type Arguments

C++ You cannot use one of the following as a template argument for a type template parameter:

- a local type
- a type with no linkage
- an unnamed type
- a type compounded from any of the above types

If it is ambiguous whether a template argument is a type or an expression, the template argument is considered to be a type. The following example demonstrates this:

```
template<class T> void f() { };
template<int i> void f() { };

int main() {
    f<int>();
}
```

The function call `f<int>()` calls the function with `T` as a template argument – the compiler considers `int()` as a type – and therefore implicitly instantiates and calls the first `f()`.

RELATED REFERENCES

- “Local Scope” on page 1
- “No Linkage” on page 7
- “Declaring and Using Bit Fields in Structures” on page 55
- “typedef” on page 43

Template Non-Type Arguments

C++ A non-type template argument provided within a template argument list is an expression whose value can be determined at compile time. Such arguments must be constant expressions, addresses of functions or objects with external linkage, or addresses of static class members. Non-type template arguments are normally used to initialize a class or to specify the sizes of class members.

For non-type integral arguments, the instance argument matches the corresponding template argument as long as the instance argument has a value and sign appropriate to the argument type.

For non-type address arguments, the type of the instance argument must be of the form `identifier` or `&identifier`, and the type of the instance argument must match the template argument exactly, except that a function name is changed to a pointer to function type before matching.

The resulting values of non-type template arguments within a template argument list form part of the template class type. If two template class names have the same template name and if their arguments have identical values, they are the same class.

In the following example, a class template is defined that requires a non-type template **int** argument as well as the type argument:

```
template<class T, int size> class myfilebuf
{
    T* filepos;
    static int array[size];
public:
    myfilebuf() { /* ... */ }
    ~myfilebuf();
    advance(); // function defined elsewhere in program
};
```

In this example, the template argument `size` becomes a part of the template class name. An object of such a template class is created with both the type argument `T` of the class and the value of the non-type template argument `size`.

An object `x`, and its corresponding template class with arguments **double** and `size=200`, can be created from this template with a value as its second template argument:

```
myfilebuf<double,200> x;
```

`x` can also be created using an arithmetic expression:

```
myfilebuf<double,10*20> x;
```

The objects created by these expressions are identical because the template arguments evaluate identically. The value `200` in the first expression could have been represented by an expression whose result at compile time is known to be equal to `200`, as shown in the second construction.

Note: Arguments that contain the `<` symbol or the `>` symbol must be enclosed in parentheses to prevent it from being parsed as a template argument list delimiter when it is being used as a relational operator or a nested template delimiter. For example, the arguments in the following definition are valid:

```
myfilebuf<double, (75>25)> x;           // valid
```

The following definition, however, is not valid because the greater than operator (`>`) is interpreted as the closing delimiter of the template argument list:

```
myfilebuf<double, 75>25> x;           // error
```

If the template arguments do not evaluate identically, the objects created are of different types:

```
myfilebuf<double,200> x; // create object x of class
                        // myfilebuf<double,200>
myfilebuf<double,200.0> y; // error, 200.0 is a double,
                        // not an int
```

The instantiation of `y` fails because the value `200.0` is of type **double**, and the template argument is of type **int**.

The following two objects:

```
myfilebuf<double, 128> x
myfilebuf<double, 512> y
```

are objects of separate template specializations. Referring either of these objects later with `myfilebuf<double>` is an error.

A class template does not need to have a type argument if it has non-type arguments. For example, the following template is a valid class template:

```
template<int i> class C
{
    public:
        int k;
        C() { k = i; }
};
```

This class template can be instantiated by declarations such as:

```
class C<100>;
class C<200>;
```

Again, these two declarations refer to distinct classes because the values of their non-type arguments differ.

RELATED REFERENCES

- “Integer Constant Expressions” on page 101
- “References” on page 92
- “External Linkage” on page 6
- “Static Members” on page 303

Template Template Arguments

> C++ A template argument for a template template parameter is the name of a class template.

When the compiler tries to find a template to match the template template argument, it only considers primary class templates. (A *primary template* is the template that is being specialized.) The compiler will not consider any partial specialization even if their parameter lists match that of the template template parameter. For example, the compiler will not allow the following code:

```
template<class T, int i> class A {
    int x;
};

template<class T> class A<T, 5> {
    short x;
};

template<template<class T> class U> class B1 { };

B1<A> c;
```

The compiler will not allow the declaration `B1<A> c`. Although the partial specialization of `A` seems to match the template template parameter `U` of `B1`, the compiler considers only the primary template of `A`, which has different template parameters than `U`.

The compiler considers the partial specializations based on a template template argument once you have instantiated a specialization based on the corresponding template template parameter. The following example demonstrates this:

```
#include <iostream>
using namespace std;

template<class T, class U> class A {
    int x;
};

template<class U> class A<int, U> {
    short x;
};

template<template<class T, class U> class V> class B {
    V<int, char> i;
    V<char, char> j;
};

B<A> c;

int main() {
    cout << typeid(c.i.x).name() << endl;
    cout << typeid(c.j.x).name() << endl;
}
```

The following is the output of the above example:

```
short
int
```

The declaration `V<int, char> i` uses the partial specialization while the declaration `V<char, char> j` uses the primary template.

RELATED REFERENCES

- “Partial Specialization” on page 395
- “Template Instantiation” on page 387

Class Templates

C++ The relationship between a class template and an individual class is like the relationship between a class and an individual object. An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated.

Note the distinction between the terms *class template* and *template class*:

Class template

is a template used to generate template classes. You cannot declare an object of a class template.

Template class

is an instance of a class template.

A template definition is identical to any valid class definition that the template might generate, except for the following:

- The class template definition is preceded by

```
template
< template-parameter-list >
```

where *template-parameter-list* is a comma-separated list of one or more of the following kinds of template parameters:

- type
 - non-type
 - template
- Types, variables, constants and objects within the class template can be declared using the template parameters as well as explicit types (for example, **int** or **char**).

A class template can be declared without being defined by using an elaborated type specifier. For example:

```
template
<class L,class T> class key;
```

This reserves the name as a class template name. All template declarations for a class template must have the same types and number of template arguments. Only one template declaration containing the class definition is allowed.

Note: When you have nested template argument lists, you must have a separating space between the > at the end of the inner list and the one at the end of the outer list. Otherwise, there is an ambiguity between the output operator >> and two template list delimiters >.

```
template <class L,class T> class key { /* ... */
};
template <class L> class vector { /* ... */ };

int main ()
{
    class key <int, vector<int> >;
    // implicitly instantiates template
}
```

Objects and function members of individual template classes can be accessed by any of the techniques used to access ordinary class member objects and functions. Given a class template:

```
template<class T> class vehicle
{
public:
    vehicle() { /* ... */ }    // constructor
    ~vehicle() {};            // destructor
    T kind[16];
    T* drive();
    static void roadmap();
    // ...
};
```

and the declaration:

```
vehicle<char> bicycle; // instantiates the template
```

the constructor, the constructed object, and the member function `drive()` can be accessed with any of the following (assuming the standard header file `<string.h>` is included in the program file):

constructor	<pre> vehicle<char> bicycle; // constructor called automatically, // object bicycle created </pre>
object bicycle	<pre> strcpy (bicycle.kind, "10 speed"); bicycle.kind[0] = '2'; </pre>
function drive()	<pre> char* n = bicycle.drive(); </pre>
function roadmap()	<pre> vehicle<char>::roadmap(); </pre>

RELATED REFERENCES

- “Declaring Class Types” on page 283
- “Scope of Class Names” on page 287

Class Template Declarations and Definitions

C++ A class template must be declared before any declaration of a corresponding template class. A class template definition can only appear once in any single compilation unit. A class template must be defined before any use of a template class that requires the size of the class or refers to members of the class.

VAC++ None of these order restrictions apply to the incremental compiler with unordered programming.

In the following example, the class template `key` is declared before it is defined. The declaration of the pointer `keyiptr` is valid because the size of the class is not needed. The declaration of `keyi`, however, causes an error.

```

template <class L> class key;           // class template declared,
                                       // not defined yet
                                       //
class key<int> *keyiptr;               // declaration of pointer
                                       //
class key<int> keyi;                   // error, cannot declare keyi
                                       // without knowing size
                                       //
template <class L> class key           // now class template defined
{ /* ... */ };

```

If a template class is used before the corresponding class template is defined, the compiler issues an error. A class name with the appearance of a template class name is considered to be a template class. In other words, angle brackets are valid in a class name only if that class is a template class.

The definition of a class template is not compiled until the definition of a template class is required. At that point, the class template definition is compiled using the argument list of the template class to instantiate the template arguments. Any errors in the class definition are flagged at this time.

z/OS In the z/OS implementation, the compiler checks the syntax of the entire template class definition when the `TEMPINC` files are being compiled if the `TEMPINC` compiler option is used, or during the original compiler pass if the `NOTEMPINC` compiler option is used. Any errors in the class definition are flagged. The compiler

does not generate code or data until it requires a specialization. At that point it generates appropriate code and data for the specialization by using the argument list supplied.

RELATED REFERENCES

- “Class Templates” on page 374

Static Data Members and Templates

C++ Each class template instantiation has its own copy of any static data members. The static declaration can be of template argument type or of any defined type.

You must separately define static members. The following example demonstrates this:

```
template <class T> class K
{
public:
    static T x;
};
template <class T> T K<T> ::x;

int main()
{
    K<int>::x = 0;
}
```

The statement `template<class T> T K<T>::x` defines the static member of class `T`, while the statement in the `main()` function initializes the data member for `K`.

RELATED REFERENCES

- “Static Members” on page 303

Member Functions of Class Templates

C++ You may define a template member function outside of its class template definition.

When you call a member function of a class template specialization, the compiler will use the template arguments that you used to generate the class template. The following example demonstrates this:

```
template<class T> class X {
public:
    T operator+(T);
};

template<class T> T X<T>::operator+(T arg1) {
    return arg1;
};

int main() {
    X<char> a;
    X<int> b;
    a + 'z';
    b + 4;
}
```

The overloaded addition operator has been defined outside of class X. The statement `a + 'z'` is equivalent to `a.operator+('z')`. The statement `b + 4` is equivalent to `b.operator+(4)`.

RELATED REFERENCES

- “Member Functions” on page 295

Friends and Templates

C++ There are four kinds of relationships between classes and their friends when templates are involved:

- *One-to-many*: A non-template function may be a friend to all template class instantiations.
- *Many-to-one*: All instantiations of a template function may be friends to a regular non-template class.
- *One-to-one*: A template function instantiated with one set of template arguments may be a friend to one template class instantiated with the same set of template arguments. This is also the relationship between a regular non-template class and a regular non-template friend function.
- *Many-to-many*: All instantiations of a template function may be a friend to all instantiations of the template class.

The following example demonstrates these relationships:

```
class B{
    template<class V> friend int j();
}
```

```
template<class S> g();
```

```
template<class T> class A {
    friend int e();
    friend int f(T);
    friend int g<T>();
    template<class U> friend int h();
};
```

- Function `e()` has a one-to-many relationship with class A. Function `e()` is a friend to all instantiations of class A.
- Function `f()` has a one-to-one relationship with class A. The compiler will give you a warning for this kind of declaration similar to the following:

The friend function declaration “f” will cause an error when the enclosing template class is instantiated with arguments that declare a friend function that does not match an existing definition. The function declares only one function because it is not a template but the function type depends on one or more template parameters.

- Function `g()` has a one-to-one relationship with class A. Function `g()` is a function template. It must be declared before here or else the compiler will not recognize `g<T>` as a template name. For each instantiation of A there is one matching instantiation of `g()`. For example, `g<int>` is a friend of `A<int>`.
- Function `h()` has a many-to-many relationship with class A. Function `h()` is a function template. For all instantiations of A all instantiations of `h()` are friends.
- Function `j()` has a many-to-one relationship with class B.

These relationships also apply to friend classes.

RELATED REFERENCES

- “Friends” on page 310

Function Templates

► C++ A *function template* defines how a group of functions can be generated.

A non-template function is not related to a function template, even though the non-template function may have the same name and parameter profile as those of a specialization generated from a template. A non-template function is never considered to be a specialization of a function template.

The following example implements the QuickSort algorithm with a function template named `quicksort`:

```
#include <iostream>
#include <cstdlib>
using namespace std;

template<class T> void quicksort(T a[], const int& leftarg, const int& rightarg) {
    if (leftarg < rightarg) {
        T pivotvalue = a[leftarg];
        int left = leftarg - 1;
        int right = rightarg + 1;

        for(;;) {
            while (a[--right] > pivotvalue);
            while (a[++left] < pivotvalue);

            if (left >= right) break;

            T temp = a[right];
            a[right] = a[left];
            a[left] = temp;
        }

        int pivot = right;
        quicksort(a, leftarg, pivot);
        quicksort(a, pivot + 1, rightarg);
    }
}

int main(void) {
    int sortme[10];

    for (int i = 0; i < 10; i++) {
        sortme[i] = rand();
        cout << sortme[i] << " ";
    };
    cout << endl;

    quicksort<int>(sortme, 0, 10 - 1);

    for (int i = 0; i < 10; i++) cout << sortme[i] << "
";
    cout << endl;
    return 0;
}
```

The above example will have output similar to the following:

```
16838 5758 10113 17515 31051 5627 23010 7419 16212 4086
4086 5627 5758 7419 10113 16212 16838 17515 23010 31051
```

This QuickSort algorithm will sort an array of type `T` (whose relational and assignment operators have been defined). The template function takes one template argument and three function arguments:

- the type of the array to be sorted, `T`
- the name of the array to be sorted, `a`
- the lower bound of the array, `leftarg`
- the upper bound of the array, `rightarg`

In the above example, you can also call the `quicksort()` template function with the following statement:

```
quicksort(sortme, 0, 10 - 1);
```

You may omit any template argument if the compiler can deduce it by the usage and context of the template function call. In this case, the compiler deduces that `sortme` is an array of type `int`.

RELATED REFERENCES

- “Template Argument Deduction”

Template Argument Deduction

C++ When you call a template function, you may omit any template argument that the compiler can determine or *deduce* by the usage and context of that template function call.

The compiler tries to deduce a template argument by comparing the type of the corresponding template parameter with the type of the argument used in the function call. The two types that the compiler compares (the template parameter and the argument used in the function call) must be of a certain structure in order for template argument deduction to work. The following lists these type structures:

```
T
const T
volatile T
T&
T*
T[10]
A<T>
C(*) (T)
T(*) ()
T(*) (U)
T C::*
C T::*
T U::*
T (C::*) ()
C (T::*) ()
D (C::*) (T)
C (T::*) (U)
T (C::*) (U)
T (U::*) ()
T (U::*) (V)
E[10][i]
B<i>
TT<T>
TT<i>
TT<C>
```

- `T`, `U`, and `V` represent a template type argument
- `10` represents any integer constant
- `i` represents a template non-type argument

- `[i]` represents an array bound of a reference or pointer type, or a non-major array bound of a normal array.
- `TT` represents a template template argument
- `(T)`, `(U)`, and `(V)` represents an argument list that has at least one template type argument
- `()` represents an argument list that has no template arguments
- `<T>` represents a template argument list that has at least one template type argument
- `<i>` represents a template argument list that has at least one template non-type argument
- `<C>` represents a template argument list that has no template arguments dependent on a template parameter

The following example demonstrates the use of each of these type structures. The example declares a template function using each of the above structures as an argument. These functions are then called (without template arguments) in order of declaration. The example outputs the same list of type structures:

```
#include <iostream>
using namespace std;

template<class T> class A { };
template<int i> class B { };

class C {
public:
    int x;
};

class D {
public:
    C y;
    int z;
};

template<class T> void f(T)           {cout << "T" << endl; };
template<class T> void f1(const T)    {cout << "const T" << endl; };
template<class T> void f2(volatile T) {cout << "volatile T" << endl; };
template<class T> void g(T*)          {cout << "T*" << endl; };
template<class T> void g(T&)           {cout << "T&" << endl; };
template<class T> void g1(T[10])      {cout << "T[10]" << endl; };
template<class T> void h1(A<T>)       {cout << "A<T>" << endl; };

void test_1() {
    A<char> a;
    C c;

    f(c);   f1(c);   f2(c);
    g(c);   g(&c);   g1(&c);
    h1(a);
}

template<class T> void j(C(*) (T)) {cout << "C(*) (T)" << endl; };
template<class T> void j(T(*) ()) {cout << "T(*) ()" << endl; };
template<class T, class U> void j(T(*) (U)) {cout << "T(*) (U)" << endl; };

void test_2() {
    C (*c_pfunct1)(int);
    C (*c_pfunct2)(void);
    int (*c_pfunct3)(int);
    j(c_pfunct1);
    j(c_pfunct2);
    j(c_pfunct3);
}
```

```

template<class T>          void k(T C::*) {cout << "T C::*" << endl; };
template<class T>          void k(C T::*) {cout << "C T::*" << endl; };
template<class T, class U> void k(T U::*) {cout << "T U::*" << endl; };

void test_3() {
    k(&C::x);
    k(&D::y);
    k(&D::z);
}

template<class T>          void m(T (C::*)() ) {cout << "T (C::*)()" << endl; };
template<class T>          void m(C (T::*)() ) {cout << "C (T::*)()" << endl; };
template<class T>          void m(D (C::*)(T)) {cout << "D (C::*)(T)" << endl; };
template<class T, class U> void m(C (T::*)(U)) {cout << "C (T::*)(U)" << endl; };
template<class T, class U> void m(T (C::*)(U)) {cout << "T (C::*)(U)" << endl; };
template<class T, class U> void m(T (U::*)() ) {cout << "T (U::*)()" << endl; };
template<class T, class U, class V> void m(T (U::*)(V)) {cout << "T (U::*)(V)" << endl; };

void test_4() {
    int (C::*f_membp1)(void);
    C (D::*f_membp2)(void);
    D (C::*f_membp3)(int);
    m(f_membp1);
    m(f_membp2);
    m(f_membp3);

    C (D::*f_membp4)(int);
    int (C::*f_membp5)(int);
    int (D::*f_membp6)(void);
    m(f_membp4);
    m(f_membp5);
    m(f_membp6);

    int (D::*f_membp7)(int);
    m(f_membp7);
}

template<int i> void n(C[10][i]) {cout << "E[10][i]" << endl; };
template<int i> void n(B<i>) {cout << "B<i>" << endl; };

void test_5() {
    C array[10][20];
    n(array);
    B<20> b;
    n(b);
}

template<template<class> class TT, class T> void p1(TT<T>) {cout << "TT<T>" << endl; };
template<template<int> class TT, int i> void p2(TT<i>) {cout << "TT<i>" << endl; };
template<template<class> class TT> void p3(TT<C>) {cout << "TT<C>" << endl; };

void test_6() {
    A<char> a;
    B<20> b;
    A<C> c;
    p1(a);
    p2(b);
    p3(c);
}

int main() { test_1(); test_2(); test_3(); test_4(); test_5(); test_6(); }

```

RELATED REFERENCES

- “Deducing Type Template Arguments” on page 383

- “Deducing Non-Type Template Arguments” on page 384

Deducing Type Template Arguments

C++ The compiler can deduce template arguments from a type composed of several of the listed type structures. The following example demonstrates template argument deduction for a type composed of several type structures:

```
template<class T> class Y { };

template<class T, int i> class X {
public:
    Y<T> f(char[20][i]) { return x; };
    Y<T> x;
};

template<template<class> class T, class U, class V, class W, int i>
    void g( T<U> (V::*)(W[20][i]) ) { };

int main()
{
    Y<int> (X<int, 20>::*p)(char[20][20]) = &X<int, 20>::f;
    g(p);
}
```

The type `Y<int> (X<int, 20>::*p)(char[20][20])T<U> (V::*)(W[20][i])` is based on the type structure `T (U::*)(V):`

- T is `Y<int>`
- U is `X<int, 20>`
- V is `char[20][20]`

If you qualify a type with the class to which that type belongs, and that class (a *nested name specifier*) depends on a template parameter, the compiler will not deduce a template argument for that parameter. If a type contains a template argument that cannot be deduced for this reason, all template arguments in that type will not be deduced. The following example demonstrates this:

```
template<class T, class U, class V>
    void h(typename Y<T>::template Z<U>, Y<T>, Y<V>) { };

int main() {
    Y<int>::Z<char> a;
    Y<int> b;
    Y<float> c;

    h<int, char, float>(a, b, c);
    h<int, char>(a, b, c);
    // h<int>(a, b, c);
}
```

The compiler will not deduce the template arguments T and U in `typename Y<T>::template Z<U>` (but it will deduce the T in `Y<T>`). The compiler would not allow the template function call `h<int>(a, b, c)` because U is not deduced by the compiler.

The compiler can deduce a function template argument from a pointer to function or pointer to member function argument given several overloaded function names. However, none of the overloaded functions may be function templates, nor can more than one overloaded function match the required type. The following example demonstrates this:

```
template<class T> void f(void(*) (T,int)) { };

template<class T> void g1(T, int) { };
```

```

void g2(int, int) { };
void g2(char, int) { };

void g3(int, int, int) { };
void g3(float, int) { };

int main() {
    // f(&g1);
    // f(&g2);
    f(&g3);
}

```

The compiler would not allow the call `f(&g1)` because `g1()` is a function template. The compiler would not allow the call `f(&g2)` because both functions named `g2()` match the type required by `f()`.

The compiler cannot deduce a template argument from the type of a default argument. The following example demonstrates this:

```

template<class T> void f(T = 2, T = 3) { };

int main() {
    f(6);
    // f();
    f<int>();
}

```

The compiler allows the call `f(6)` because the compiler deduces the template argument (`int`) by the value of the function call's argument. The compiler would not allow the call `f()` because the compiler cannot deduce template argument from the default arguments of `f()`.

The compiler cannot deduce a template type argument from the type of a non-type template argument. For example, the compiler will not allow the following:

```

template<class T, T i> void f(int[20][i]) { };

int main() {
    int a[20][30];
    f(a);
}

```

The compiler cannot deduce the type of template parameter `T`.

RELATED REFERENCES

- “Template Argument Deduction” on page 380

Deducing Non-Type Template Arguments

C++ The compiler cannot deduce the value of a major array bound unless the bound refers to a reference or pointer type. Major array bounds are not part of function parameter types. The following code demonstrates this:

```

template<int i> void f(int a[10][i]) { };
template<int i> void g(int a[i]) { };
template<int i> void h(int (&a)[i]) { };

int main () {
    int b[10][20];
    int c[10];
}

```

```

    f(b);
    // g(c);
    h(c);
}

```

The compiler would not allow the call `g(c)`; the compiler cannot deduce template argument `i`.

The compiler cannot deduce the value of a non-type template argument used in an expression in the template function's parameter list. The following example demonstrates this:

```

template<int i> class X { };

template<int i> void f(X<i - 1>) { };

int main () {
    X<0> a;
    f<1>(a);
    // f(a);
}

```

In order to call function `f()` with object `a`, the function must accept an argument of type `X<0>`. However, the compiler cannot deduce that the template argument `i` must be equal to 1 in order for the function template argument type `X<i - 1>` to be equivalent to `X<0>`. Therefore the compiler would not allow the function call `f(a)`.

If you want the compiler to deduce a non-type template argument, the type of the parameter must match exactly the type of value used in the function call. For example, the compiler will not allow the following:

```

template<int i> class A { };
template<short d> void f(A<d>) { };

int main() {
    A<1> a;
    f(a);
}

```

The compiler will not convert `int` to `short` when the example calls `f()`.

However, deduced array bounds may be of any integral type.

RELATED REFERENCES

- “Template Argument Deduction” on page 380

Overloading Function Templates

C++ You may overload a function template either by a non-template function or by another function template.

If you call the name of an overloaded function template, the compiler will try to deduce its template arguments and check its explicitly declared template arguments. If successful, it will instantiate a function template specialization, then add this specialization to the set of *candidate functions* used in overload resolution. The compiler proceeds with overload resolution, choosing the most appropriate function from the set of candidate functions. Non-template functions take precedence over template functions. The following example describes this:

```
#include <iostream>
using namespace std;

template<class T> void f(T x, T y) { cout << "Template" << endl; }

void f(int w, int z) { cout << "Non-template" << endl; }

int main() {
    f( 1, 2 );
    f('a', 'b');
    f( 1, 'b');
}
```

The following is the output of the above example:

```
Non-template
Template
Non-template
```

The function call `f(1, 2)` could match the argument types of both the template function and the non-template function. Because in overload resolution non-template functions take precedence, the non-template function is called.

The function call `f('a', 'b')` can only match the argument types of the template function. The template function is called.

Argument deduction fails for the function call `f(1, 'b')`; the compiler does not generate any template function specialization and overload resolution does not take place. The non-template function resolves this function call after using the standard conversion from **char** to **int** for the function argument `'b'`.

RELATED REFERENCES

- “Overload Resolution” on page 280

Partial Ordering of Function Templates

C++ A function template specialization might be ambiguous because template argument deduction might associate the specialization with more than one of the overloaded definitions. The compiler will then choose the definition that is the most specialized. This process of selecting a function template definition is called *partial ordering*.

A template *X* is more specialized than a template *Y* if every argument list that matches the one specified by *X* also matches the one specified by *Y*, but not the other way around. The following example demonstrates partial ordering:

```
template<class T> void f(T) { }
template<class T> void f(T*) { }
template<class T> void f(const T*) { }

template<class T> void g(T) { }
template<class T> void g(T&) { }

template<class T> void h(T) { }
template<class T> void h(T, ...) { }

int main() {
    const int *p;
    f(p);
}
```

```

    int q;
    // g(q);
    // h(q);
}

```

The declaration `template<class T> void f(const T*)` is more specialized than `template<class T> void f(T*)`. Therefore, the function call `f(p)` calls `template<class T> void f(const T*)`. However, neither `void g(T)` nor `void g(T&)` is more specialized than the other. Therefore, the function call `g(q)` would be ambiguous.

Ellipses do not affect partial ordering. Therefore, the function call `h(q)` would also be ambiguous.

The compiler uses partial ordering in the following cases:

- Calling a function template specialization that requires overload resolution.
- Taking the address of a function template specialization.
- When a friend function declaration, an explicit instantiation, or explicit specialization refers to a function template specialization.
- Determining the appropriate deallocation function that is also a function template for a given placement operator `new`.

RELATED REFERENCES

- “Template Specialization” on page 390
- “C++ new Operator” on page 119

Template Instantiation

C++ The act of creating a new definition of a function, class, or member of a class from a template declaration and one or more template arguments is called *template instantiation*. The definition created from a template instantiation is called a *specialization*.

RELATED REFERENCES

- “Template Specialization” on page 390

Implicit Instantiation

C++ Unless a template specialization has been explicitly instantiated or explicitly specialized, the compiler will generate a specialization for the template only when it needs the definition. This is called *implicit instantiation*.

If the compiler must instantiate a class template specialization and the template is declared, you must also define the template.

For example, if you declare a pointer to a class, the definition of that class is not needed and the class will not be implicitly instantiated. The following example demonstrates when the compiler instantiates a template class:

```

template<class T> class X {
public:
    X* p;
    void f();
    void g();
};

```

```

X<int>* q;
X<int> r;
X<float>* s;
r.f();
s->g();

```

The compiler requires the instantiation of the following classes and functions:

- X<int> when the object r is declared
- X<int>::f() at the member function call r.f()
- X<float> and X<float>::g() at the class member access function call s->g()

Therefore, the functions X<T>::f() and X<T>::g() must be defined in order for the above example to compile. (The compiler will use the default constructor of class X when it creates object r.) The compiler does not require the instantiation of the following definitions:

- class X when the pointer p is declared
- X<int> when the pointer q is declared
- X<float> when the pointer s is declared

The compiler will implicitly instantiate a class template specialization if it is involved in pointer conversion or pointer to member conversion. The following example demonstrates this:

```

template<class T> class B { };
template<class T> class D : public B<T> { };

void g(D<double>* p, D<int>* q)
{
    B<double>* r = p;
    delete q;
}

```

The assignment B<double>* r = p converts p of type D<double>* to a type of B<double>*; the compiler must instantiate D<double>. The compiler must instantiate D<int> when it tries to delete q.

If the compiler implicitly instantiates a class template that contains static members, those static members are not implicitly instantiated. The compiler will instantiate a static member only when the compiler needs the static member's definition. Every instantiated class template specialization has its own copy of static members. The following example demonstrates this:

```

template<class T> class X {
public:
    static T v;
};

template<class T> T X<T>::v = 0;

X<char*> a;
X<float> b;
X<float> c;

```

Object a has a static member variable v of type char*. Object b has a static variable v of type float. Objects b and c share the single static data member v.

An implicitly instantiated template is in the same namespace where you defined the template.

If a function template or a member function template specialization is involved with overload resolution, the compiler implicitly instantiates a declaration of the specialization.

RELATED REFERENCES

- “Template Instantiation” on page 387

Explicit Instantiation

C++ You can explicitly tell the compiler when it should generate a definition from a template. This is called *explicit instantiation*.

Syntax — Explicit Instantiation Declaration

►—template—*template_declaration*—◄◄

The following are examples of explicit instantiations:

```
template<class T> class Array { void mf(); };
template class Array<char>;           // explicit instantiation
template void Array<int>::mf();       // explicit instantiation

template<class T> void sort(Array<T>& v) { }
template void sort(Array<char>&);     // explicit instantiation

namespace N {
    template<class T> void f(T&) { }
}

template void N::f<int>(int&);
// The explicit instantiation is in namespace N.

int* p = 0;
template<class T> T g(T = &p);
template char g(char);                // explicit instantiation

template <class T> class X {
    private:
        T v(T arg) { return arg; };
};

template int X<int>::v(int);           // explicit instantiation

template<class T> T g(T val) {return val;}
template<class T> void Array<T>::mf() { }
```

A template declaration must be in scope at the point of instantiation of the template’s explicit instantiation. An explicit instantiation of a template specialization is in the same namespace where you defined the template.

Access checking rules do not apply to names in explicit instantiations. Template arguments and names in a declaration of an explicit instantiation may be private types or objects. In the above example, the compiler allows the explicit instantiation `template int X<int>::v(int)` even though the member function is declared private.

The compiler does not use default arguments when you explicitly instantiate a template. In the above example the compiler allows the explicit instantiation `template char g(char)` even though the default argument is an address of type `int`.

RELATED REFERENCES

- “Template Instantiation” on page 387

Template Specialization

C++ The act of creating a new definition of a function, class, or member of a class from a template declaration and one or more template arguments is called *template instantiation*. The definition created from a template instantiation is called a *specialization*. A *primary template* is the template that is being specialized.

RELATED REFERENCES

- “Template Instantiation” on page 387

Explicit Specialization

C++ When you instantiate a template with a given set of template arguments the compiler generates a new definition based on those template arguments. You can override this behavior of definition generation. You can instead specify the definition the compiler uses for a given set of template arguments. This is called *explicit specialization*. You can explicitly specialize any of the following:

- Function or class template
- Member function of a class template
- Static data member of a class template
- Member class of a class template
- Member function template of a class template
- Member class template of a class template

Syntax — Explicit Specialization Declaration

►►—template—<—>—*declaration_name*—<—*template_argument_list*—>—*declaration_body*—◄◄

The **template<>** prefix indicates that the following template declaration takes no template parameters. The *declaration_name* is the name of a previously declared template. Note that you can forward-declare an explicit specialization so the *declaration_body* is optional, at least until the specialization is referenced.

The following example demonstrates explicit specialization:

```
using namespace std;

template<class T = float, int i = 5> class A
{
    public:
        A();
        int value;
};

template<> class A<> { public: A(); };
template<> class A<double, 10> { public: A(); };

template<class T, int i> A<T, i>::A() : value(i) {
    cout << "Primary template, "
        << "non-type argument is " << value << endl;
}

A<>::A() {
    cout << "Explicit specialization "
        << "default arguments" << endl;
}
```



```

A<double, 10>::A() {
    cout << "Explicit specialization "
          << "<double, 10>" << endl;
}

int main() {
    A<int, 6> x;
    A<> y;
    A<double, 10> z;
}

```

The following is the output of the above example:

```

Primary template non-type argument is: 6
Explicit specialization default arguments
Explicit specialization <double, 10>

```

This example declared two explicit specializations for the *primary template* (the template which is being specialized) class A. Object x uses the constructor of the primary template. Object y uses the explicit specialization A<>::A(). Object z uses the explicit specialization A<double, 10>::A().

RELATED REFERENCES

- “Function Templates” on page 379
- “Class Templates” on page 374
- “Member Functions of Class Templates” on page 377
- “Static Data Members and Templates” on page 377

Definition and Declaration of Explicit Specializations

C++ The definition of an explicitly specialized class is unrelated to the definition of the primary template. You do not have to define the primary template in order to define the specialization (nor do you have to define the specialization in order to define the primary template). For example, the compiler will allow the following:

```

template<class T> class A;
template<> class A<int>;

template<> class A<int> { /* ... */ };

```

The primary template is not defined, but the explicit specialization is.

You can use the name of an explicit specialization that has been declared but not defined the same way as an incompletely defined class. The following example demonstrates this:

```

template<class T> class X { };
template<> class X<char>;
X<char>* p;
X<int> i;
// X<char> j;

```

The compiler does not allow the declaration X<char> j because the explicit specialization of X<char> is not defined.

RELATED REFERENCES

- “Explicit Specialization” on page 390

Explicit Specialization and Scope

► C++ A declaration of a primary template must be in scope at the *point of declaration* of the explicit specialization. In other words, an explicit specialization declaration must appear after the declaration of the primary template. For example, the compiler will not allow the following:

```
template<> class A<int>;
template<class T> class A;
```

An explicit specialization is in the same namespace as the definition of the primary template.

RELATED REFERENCES

- “Explicit Specialization” on page 390

Class Members of Explicit Specializations

► C++ A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the primary template. You have to explicitly define members of a class template specialization. You define members of an explicitly specialized template class as you would normal classes, without the **template<>** prefix. In addition, you can define the members of an explicit specialization inline; no special template syntax is used in this case. The following example demonstrates a class template specialization:

```
template<class T> class A {
    public:
        void f(T);
};

template<> class A<int> {
    public:
        int g(int);
};

int A<int>::g(int arg) { return 0; }

int main() {
    A<int> a;
    a.g(1234);
}
```

The explicit specialization `A<int>` contains the member function `g()`, which the primary template does not.

If you explicitly specialize a template, a member template, or the member of a class template, then you must declare this specialization before that specialization is implicitly instantiated. For example, the compiler will not allow the following code:

```
template<class T> class A { };

void f() { A<int> x; }
template<> class A<int> { };

int main() { f(); }
```

The compiler will not allow the explicit specialization `template<> class A<int> { };` because function `f()` uses this specialization (in the construction of `x`) before the specialization.

>VAC++ The unordered incremental compiler allows the above example. In this case the compiler will use the explicit specialization when it calls function `f()`.

RELATED REFERENCES

- “Explicit Specialization” on page 390

Explicit Specialization of Function Templates

> C++ In a function template specialization, a template argument is optional if the compiler can deduce it from the type of the function arguments. The following example demonstrates this:

```
template<class T> class X { };
template<class T> void f(X<T>);
template<> void f(X<int>);
```

The explicit specialization `template<> void f(X<int>)` is equivalent to `template<> void f<int>(X<int>)`.

You cannot specify default function arguments in a declaration or a definition for any of the following:

- Explicit specialization of a function template
- Explicit specialization of a member function template

For example, the compiler will not allow the following code:

```
template<class T> void f(T a) { };
template<> void f<int>(int a = 5) { };

template<class T> class X {
    void f(T a) { }
};
template<> void X<int>::f(int a = 10) { };
```

RELATED REFERENCES

- “Explicit Specialization” on page 390
- “Function Templates” on page 379

Explicit Specialization of Members of Class Templates

> C++ Each instantiated class template specialization has its own copy of any static members. You may explicitly specialize static members. The following example demonstrates this:

```
template<class T> class X {
public:
    static T v;
    static void f(T);
};

template<class T> T X<T>::v = 0;
template<class T> void X<T>::f(T arg) { v = arg; }

template<> char* X<char*>::v = "Hello";
template<> void X<float>::f(float arg) { v = arg * 2; }

int main() {
    X<char*> a, b;
    X<float> c;
    c.f(10);
}
```

This code explicitly specializes the initialization of static data member `X::v` to point to the string "Hello" for the template argument `char*`. The function `X::f()` is explicitly specialized for the template argument `float`. The static data member `v` in objects `a` and `b` point to the same string, "Hello". The value of `c.v` is equal to 20 after the call function call `c.f(10)`.

You can nest member templates within many enclosing class templates. If you explicitly specialize a template nested within several enclosing class templates, you must prefix the declaration with `template<>` for every enclosing class template you specialize. You may leave some enclosing class templates unspecialized, however you cannot explicitly specialize a class template unless its enclosing class templates are also explicitly specialized. The following example demonstrates explicit specialization of nested member templates:

```
#include <iostream>
using namespace std;

template<class T> class X {
public:
    template<class U> class Y {
    public:
        template<class V> void f(U,V);
        void g(U);
    };
};

template<class T> template<class U> template<class V>
void X<T>::Y<U>::f(U, V) { cout << "Template 1" << endl; }

template<class T> template<class U>
void X<T>::Y<U>::g(U) { cout << "Template 2" << endl; }

template<> template<>
void X<int>::Y<int>::g(int) { cout << "Template 3" << endl; }

template<> template<> template<class V>
void X<int>::Y<int>::f(int, V) {cout << "Template 4" << endl; }

template<> template<> template<>
void X<int>::Y<int>::f<int>(int, int) { cout << "Template 5" << endl; }

// template<> template<class U> template<class V>
// void X<char>::Y<U>::f(U, V) { cout << "Template 6" << endl; }

// template<class T> template<>
// void X<T>::Y<float>::g(float) { cout << "Template 7" << endl; }

int main() {
    X<int>::Y<int> a;
    X<char>::Y<char> b;
    a.f(1, 2);
    a.f(3, 'x');
    a.g(3);
    b.f('x', 'y');
    b.g('z');
}
```

The following is the output of the above program:

```
Template 5
Template 4
Template 3
Template 1
Template 2
```

- The compiler would not allow the template specialization definition that would output "Template 6" because it is attempting to specialize a member (function `f()`) without specialization its containing class (`Y`).
- The compiler would not allow the template specialization definition that would output "Template 7" because the enclosing class of class `Y` (which is class `X`) is not explicitly specialized.

A friend declaration cannot declare an explicit specialization.

RELATED REFERENCES

- "Explicit Specialization" on page 390
- "Static Data Members and Templates" on page 377

Partial Specialization

C++ When you instantiate a class template, the compiler creates a definition based on the template arguments you have passed. Alternatively, if all those template arguments match those of an explicit specialization, the compiler uses the definition defined by the explicit specialization.

A *partial specialization* is a generalization of explicit specialization. An explicit specialization only has a template argument list. A partial specialization has both a template argument list and a template parameter list. The compiler uses the partial specialization if its template argument list matches a subset of the template arguments of a template instantiation. The compiler will then generate a new definition from the partial specialization with the rest of the unmatched template arguments of the template instantiation.

You cannot partially specialize function templates.

Syntax — Partial Specialization

►►—template—<—template_parameter_list—>—declaration_name—<—template_argument_list—>—declaration_body—►◄

The *declaration_name* is a name of a previously declared template. Note that you can forward-declare a partial specialization so that the *declaration_body* is optional.

The following demonstrates the use of partial specializations:

```
#include <iostream>
using namespace std;

template<class T, class U, int I> struct X
{ void f() { cout << "Primary template" << endl; } };

template<class T, int I> struct X<T, T*, I>
{ void f() { cout << "Partial specialization 1" << endl; } };

template<class T, class U, int I> struct X<T*, U, I>
{ void f() { cout << "Partial specialization 2" << endl; } };

template<class T> struct X<int, T*, 10>
{ void f() { cout << "Partial specialization 3" << endl; } };

template<class T, class U, int I> struct X<T, U*, I>
```

```

    { void f() { cout << "Partial specialization 4" << endl;
    } };

int main() {
    X<int, int, 10> a;
    X<int, int*, 5> b;
    X<int*, float, 10> c;
    X<int, char*, 10> d;
    X<float, int*, 10> e;
    // X<int, int*, 10> f;
    a.f(); b.f(); c.f(); d.f(); e.f();
}

```

The following is the output of the above example:

```

Primary template
Partial specialization 1
Partial specialization 2
Partial specialization 3
Partial specialization 4

```

The compiler would not allow the declaration `X<int, int*, 10> f` because it can match template struct `X<T, T*, I>`, template struct `X<int, T*, 10>`, or template struct `X<T, U*, I>`, and none of these declarations are a better match than the others.

Each class template partial specialization is a separate template. You must provide definitions for each member of a class template partial specialization.

RELATED REFERENCES

- “Template Specialization” on page 390

Template Parameter and Argument Lists of Partial Specializations

C++ Primary templates do not have template argument lists; this list is implied in the template parameter list.

Template parameters specified in a primary template but not used in a partial specialization are omitted from the template parameter list of the partial specialization. The order of a partial specialization’s argument list is the same as the order of the primary template’s implied argument list.

In a template argument list of a partial template parameter, you cannot have an expression that involves non-type arguments unless that expression is only an identifier. In the following example, the compiler will not allow the first partial specialization, but will allow the second one:

```

template<int I, int J> class X { };

// Invalid partial specialization
template<int I> class <I * 4, I + 3> { };

// Valid partial specialization
template <int I> class <I, I> { };

```

The type of a non-type template argument cannot depend on a template parameter of a partial specialization. The compiler will not allow the following partial specialization:

```
template<class T, T i> class X { };

// Invalid partial specialization
template<class T> class X<T, 25> { };
```

A partial specialization's template argument list cannot be the same as the list implied by the primary template.

You cannot have default values in the template parameter list of a partial specialization.

RELATED REFERENCES

- "Template Parameters" on page 368
- "Template Arguments" on page 370

Matching of Class Template Partial Specializations

C++ The compiler determines whether to use the primary template or one of its partial specializations by matching the template arguments of the class template specialization with the template argument lists of the primary template and the partial specializations:

- If the compiler finds only one specialization, then the compiler generates a definition from that specialization.
- If the compiler finds more than one specialization, then the compiler tries to determine which of the specializations is the most specialized. A template *X* is more specialized than a template *Y* if every argument list that matches the one specified by *X* also matches the one specified by *Y*, but not the other way around. If the compiler cannot find the most specialized specialization, then the use of the class template is ambiguous; the compiler will not allow the program.
- If the compiler does not find any matches, then the compiler generates a definition from the primary template.

RELATED REFERENCES

- "Partial Specialization" on page 395

Name Binding and Dependent Names

C++ *Name binding* is the process of finding the declaration for each name that is explicitly or implicitly used in a template. The compiler may bind a name in the definition of a template, or it may bind a name at the instantiation of a template.

A *dependent name* is a name that depends on the type or the value of a template parameter. For example:

```
template<class T> class U : A<T>
{
    typename T::B x;
    void f(A<T>& y)
    {
        *y++;
    }
};
```

The dependent names in this example are the base class *A<T>*, the type name *T::B*, and the variable *y*.

The compiler binds dependent names when a template is instantiated. The compiler binds non-dependent names when a template is defined. For example:

```
void f(double) { cout << "Function f(double)" << endl; }

template<class T> void g(T a) {
    f(123);
    h(a);
}

void f(int) { cout << "Function f(int)" << endl; }
void h(double) { cout << "Function h(double)" << endl; }

void i() {
    extern void h(int);
    g<int>(234);
}

void h(int) { cout << "Function h(int)" << endl; }
```

The following is the output if you call function `i()`:

```
Function f(double)
Function h(double)
```

VAC++ The unordered incremental compiler will not give this result; the compiler will call the best match of the overloaded functions regardless of their location.

The *point of definition* of a template is located immediately before its definition. In this example, the point of definition of the function template `g(T)` is located immediately before the keyword **template**. Because the function call `f(123)` does not depend on a template argument, the compiler will consider names declared before the definition of function template `g(T)`. Therefore, the call `f(123)` will call `f(double)`. Although `f(int)` is a better match, it is not in scope at the point of definition of `g(T)`.

The *point of instantiation* of a template is located immediately before the declaration that encloses its use. In this example, the point of instantiation of the call to `g<int>(234)` is located immediately before `i()`. Because the function call `h(a)` depends on a template argument, the compiler will consider names declared before the instantiation of function template `g(T)`. Therefore, the call `h(a)` will call `h(double)`. It will not consider `h(int)`, because this function was not in scope at the point of instantiation of `g<int>(234)`.

Point of instantiation binding implies the following:

- A template parameter cannot depend on any local name or class member.
- An unqualified name in a template cannot depend on a local name or class member.

RELATED REFERENCES

- "Template Instantiation" on page 387

The Keyword `typename`

C++ Use the keyword **typename** if you have a qualified name that refers to a type and depends on a template parameter. Only use the keyword **typename** in template declarations and definitions. The following example illustrates the use of the keyword **typename**:


```
template<class T> class A
{
    T::x(y);
    typedef char C;
    A::C d;
}
```

The statement `T::x(y)` is ambiguous. It could be a call to function `x()` with a nonlocal argument `y`, or it could be a declaration of variable `y` with type `T::x`. C++ will interpret this statement as a function call. In order for the compiler to interpret this statement as a declaration, you would add the keyword **typename** to the beginning of it. The statement `A::C d;` is ill-formed. The class `A` also refers to `A<T>` and thus depends on a template parameter. You must add the keyword **typename** to the beginning of this declaration:

```
typename A::C d;
```

You can also use the keyword **typename** in place of the keyword **class** in template parameter declarations.

RELATED REFERENCES

- “Template Parameters” on page 368

The Keyword `template` as Qualifier

C++ Use the keyword **template** as a qualifier to distinguish member templates from other names. The following example illustrates when you must use **template** as a qualifier:

```
class A
{
    public:
        template<class T> T function_m() { };
};

template<class U> void function_n(U argument)
{
    char object_x = argument.function_m<char>();
}
```

The declaration `char object_x = argument.function_m<char>();` is ill-formed. The compiler assumes that the `<` is a less-than operator. In order for the compiler to recognize the function template call, you must add the **template** quantifier:

```
char object_x = argument.template function_m<char>();
```

If the name of a member template specialization appears after a `.`, `->`, or `::` operator, and that name has explicitly qualified template parameters, prefix the member template name with the keyword **template**. The following example demonstrates this use of the keyword **template**:

```
#include <iostream>
using namespace std;

class X {
    public:
        template <int j> struct S {
            void h() {
                cout << "member template's member function: " << j << endl;
            }
        };
        template <int i> void f() {
```

```

        cout << "Primary: " << i << endl;
    }
};

template<> void X::f<20>() {
    cout << "Specialized, non-type argument = 20" << endl;
}

template<class T> void g(T* p) {
    p->template f<100>();
    p->template f<20>();
    typename T::template S<40> s; // use of scope operator on a member template
    s.h();
}

int main()
{
    X temp;
    g(&temp);
}

```

The following is the output of the above example:

```

Primary: 100
Specialized, non-type argument = 20
member template's member function: 40

```

If you do not use the keyword **template** in these cases, the compiler will interpret the < as a less-than operator. For example, the following line of code is ill-formed:

```
p->f<100>();
```

The compiler interprets `f` as a non-template member, and the < as a less-than operator.

RELATED REFERENCES

- “Chapter 16. Templates” on page 367

Chapter 17. Exception Handling

➤ **C++** *Exception handling* is a mechanism that separates code that detects and handles exceptional circumstances from the rest of your program. Note that an exceptional circumstance is not necessarily an error.

When a function detects an exceptional situation, you represent this with an object. This object is called an *exception object*. In order to deal with the exceptional situation you *throw the exception*. This passes control, as well as the exception, to a designated block of code in a direct or indirect caller of the function that threw the exception. This block of code is called a *handler*. In a handler, you specify the types of exceptions that it may process. The C++ run time, together with the generated code, will pass control to the first appropriate handler that is able to process the exception thrown. When this happens, an exception is *caught*. A handler may *rethrow* an exception so it can be caught by another handler.

The exception handling mechanism is made up of the following elements:

- **try** blocks: a block of code that may throw an exception that you want to handle with special processing
- **catch** blocks or handlers: a block of code that is executed when a try block encounters an exception
- **throw** expression: indicates when your program encounters an exception
- exception specifications: specify which exceptions (if any) a function may throw
- **unexpected()** function: called when a function throws an exception not specified by an exception specification
- **terminate()** function: called for exceptions that are not caught

RELATED REFERENCES

- "The try Keyword"
- "catch Blocks" on page 403
- "The throw Expression" on page 409
- "Exception Specifications" on page 412
- "unexpected()" on page 415
- "terminate()" on page 416

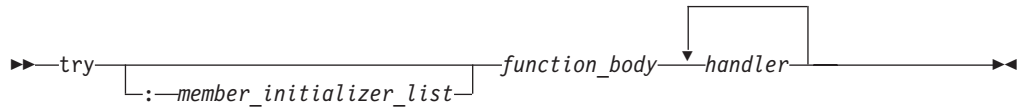
The try Keyword

➤ **C++** You use a *try block* to indicate which areas in your program that might throw exceptions you want to handle immediately. You use a *function try block* to indicate that you want to detect exceptions in the entire body of a function.

Syntax — try Block



Syntax — Function try Block



The following is an example of a function try block with a member initializer, a function try block and a try block:

```
#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { }
};

class A {
public:
    int i;

    // A function try block with a member
    // initializer
    A() try : i(0) {
        throw E("Exception thrown in A()");
    }
    catch (E& e) {
        cout << e.error << endl;
    }
};

// A function try block
void f() try {
    throw E("Exception thrown in f()");
}
catch (E& e) {
    cout << e.error << endl;
}

void g() {
    throw E("Exception thrown in g()");
}

int main() {
    f();

    // A try block
    try {
        g();
    }
    catch (E& e) {
        cout << e.error << endl;
    }
    A x;
}
```

The following is the output of the above example:

```
Exception thrown in f()
Exception thrown in g()
Exception thrown in A()
```

The constructor of class A has a function try block with a member initializer. Function f() has a function try block. The main() function contains a try block.

RELATED REFERENCES

- “Initializing Base Classes and Members” on page 346

Nested Try Blocks

C++ When try blocks are nested and a **throw** occurs in a function called by an inner try block, control is transferred outward through the nested try blocks until the first catch block is found whose argument matches the argument of the throw expression.

For example:

```
try
{
    func1();
    try
    {
        func2();
    }
    catch (spec_err) { /* ... */ }
    func3();
}
catch (type_err) { /* ... */ }
// if no throw is issued, control resumes here.
```

In the above example, if `spec_err` is thrown within the inner try block (in this case, from `func2()`), the exception is caught by the inner catch block, and, assuming this catch block does not transfer control, `func3()` is called. If `spec_err` is thrown after the inner try block (for instance, by `func3()`), it is not caught and the function **terminate()** is called. If the exception thrown from `func2()` in the inner try block is `type_err`, the program skips out of both try blocks to the second catch block without invoking `func3()`, because no appropriate catch block exists following the inner try block.

You can also nest a try block within a catch block.

RELATED REFERENCES

- “`terminate()`” on page 416
- “`unexpected()`” on page 415
- “Special Exception Handling Functions” on page 415

catch Blocks

C++ The following is the syntax for an exception handler or a catch block:

►►—catch—(—*exception_declaration*—)—{—*statements*—}—►►

You can declare a handler to catch many types of exceptions. The allowable objects that a function can catch are declared in the parentheses following the **catch** keyword (the *exception_declaration*). You can catch objects of the fundamental types, base and derived class objects, references, and pointers to all of these types. You can also catch **const** and **volatile** types. The *exception_declaration* cannot be an incomplete type, or a reference or pointer to an incomplete type other than one of the following:

- **void***

- **const void***
- **volatile void***
- **const volatile void***

You cannot define a type in an *exception_declaration*.

You can also use the **catch(...)** form of the handler to catch all thrown exceptions that have not been caught by a previous catch block. The ellipsis in the catch argument indicates that any exception thrown can be handled by this handler.

If an exception is caught by a **catch(...)** block, there is no direct way to access the object thrown. Information about an exception caught by **catch(...)** is very limited.

You can declare an optional variable name if you want to access the thrown object in the catch block.

A catch block can only catch accessible objects. The object caught must have an accessible copy constructor.

RELATED REFERENCES

- “volatile and const Qualifiers” on page 69
- “Member Access” on page 308

Function try block Handlers

C++ The scope and lifetime of the parameters of a function or constructor extend into the handlers of a function try block. The following example demonstrates this:

```
void f(int &x) try {
    throw 10;
}
catch (const int &i)
{
    x = i;
}

int main() {
    int v = 0;
    f(v);
}
```

The value of `v` after `f()` is called is 10.

A function try block on **main()** does not catch exceptions thrown in destructors of objects with static storage duration, or constructors of namespace scope objects.

The following example throws an exception from a destructor of a static object:

```
#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { }
};

class A {
public:
    ~A() { throw E("Exception in ~A()"); }
};
```

```

class B {
public: ~B() { throw E("Exception in ~B()"); }
};

int main() try {
    cout << "In main" << endl;
    static A cow;
    B bull;
}
catch (E& e) {
    cout << e.error << endl;
}

```

The following is the output of the above example:

```

In main
Exception in ~B()

```

The run time will not catch the exception thrown when object `cow` is destroyed at the end of the program.

The following example throws an exception from a constructor of a namespace scope object:

```

#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { }
};

namespace N {
    class C {
    public:
        C() {
            cout << "In C()" << endl;
            throw E("Exception in C()");
        }
    };

    C calf;
};

int main() try {
    cout << "In main" << endl;
}
catch (E& e) {
    cout << e.error << endl;
}

```

The following is the output of the above example:

```

In C()

```

The compiler will not catch the exception thrown when object `calf` is created.

In a function try block's handler, you cannot have a jump into the body of a constructor or destructor.

A return statement cannot appear in a function try block's handler of a constructor.

When the function try block's handler of an object's constructor or destructor is entered, fully constructed base classes and members of that object are destroyed. The following example demonstrates this:

```
#include
<iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { };
};

class B {
public:
    B() { };
    ~B() { cout << "~B() called" << endl; };
};

class D : public B {
public:
    D();
    ~D() { cout << "~D() called" << endl; };
};

D::~D() try : B() {
    throw E("Exception in D()");
}
catch(E& e) {
    cout << "Handler of function try block of D(): " << e.error << endl;
};

int main() {
    D val;
};
```

The following is the output of the above example:

```
~B() called
Handler of function try block of D(): Exception in D()
```

When the function try block's handler of `D()` is entered, the run time first calls the destructor of the base class of `D`, which is `B`. The destructor of `D` is not called because `val` is not fully constructed.

The run time will rethrow an exception at the end of a function try block's handler of a constructor or destructor. All other functions will return once they have reached the end of their function try block's handler. The following example demonstrates this:

```
#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { };
};

class A {
public:
    A() try { throw E("Exception in A()"); }
    catch(E& e) { cout << "Handler in A(): " << e.error << endl; }
};
```



```

int f() try {
    throw E("Exception in f()");
    return 0;
}
catch(E& e) {
    cout << "Handler in f(): " << e.error << endl;
    return 1;
}

int main() {
    int i = 0;
    try { A cow; }
    catch(E& e) {
        cout << "Handler in main(): " << e.error << endl;
    }

    try { i = f(); }
    catch(E& e) {
        cout << "Another handler in main(): " << e.error << endl;
    }

    cout << "Returned value of f(): " << i << endl;
}

```

The following is the output of the above example:

```

Handler in A(): Exception in A()
Handler in main(): Exception in A()
Handler in f(): Exception in f()
Returned value of f(): 1

```

RELATED REFERENCES

- “The main() Function” on page 162
- “static Storage Class Specifier” on page 42
- “Chapter 10. Namespaces” on page 261
- “Destructors” on page 350

Arguments of catch Blocks

► C++ If you specify a class type for the argument of a catch block (the *exception_declaration*), the compiler will use a copy constructor to initialize that argument. If that argument does not have a name, the compiler initializes a temporary object. The compiler destroys this object when the handler exits.

The ISO C++ definition permits an implementation that eliminates the construction of such temporary objects in cases in which they are redundant.

► VAC++ The VisualAge C++ compiler takes advantage of this fact to create more efficient optimized code. Take this into consideration when debugging your programs, especially for memory problems.

RELATED REFERENCES

- “Temporary Objects” on page 357

Matching between Exceptions Thrown and Caught

► C++ An argument in the catch argument of a handler matches an argument in the *assignment_expression* of the throw expression (throw argument) if any of the following conditions is met:

- The catch argument type matches the type of the thrown object.
- The catch argument is a public base class of the thrown class object.
- The catch specifies a pointer type, and the thrown object is a pointer type that can be converted to the pointer type of the catch argument by standard pointer conversion.

Note: If the type of the thrown object is **const** or **volatile**, the catch argument must also be a **const** or **volatile** for a match to occur. However, a **const**, **volatile**, or reference type catch argument can match a nonconstant, nonvolatile, or nonreference object type. A nonreference catch argument type matches a reference to an object of the same type.

RELATED REFERENCES

- “Pointer Conversions” on page 146
- “volatile and const Qualifiers” on page 69
- “References” on page 92
- “Special Exception Handling Functions” on page 415

Order of Catching

C++ If the compiler encounters an exception in a try block, it will try each handler in order of appearance.

Always place a catch block that catches a derived class before a catch block that catches the base class of that derived class (following a try block). If a catch block for objects of a base class is followed by a catch block for objects of a derived class of that base class, the compiler issues a warning and continues to compile the program despite the unreachable code in the derived class handler.

A catch block of the form **catch(...)** must be the last catch block following a try block or an error occurs. This placement ensures that the **catch(...)** block does not prevent more specific catch blocks from catching exceptions intended for them.

If the run time cannot find a matching handler in the current scope, the run time will continue to find a matching handler in a dynamically surrounding try block. The following example demonstrates this:

```
#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { };
};

class F : public E {
public:
    F(const char* arg) : E(arg) { };
};

void f() {
    try {
        cout << "In try block of f()" << endl;
        throw E("Class E exception");
    }
    catch (F& e) {
        cout << "In handler of f()";
        cout << e.error << endl;
    }
}
```

```
};

int main() {
    try {
        cout << "In main" << endl;
        f();
    }
    catch (E& e) {
        cout << "In handler of f(): ";
        cout << e.error << endl;
    };
    cout << "Resume execution in main" << endl;
}
```

The following is the output of the above example:

```
In try block of main
In try block of f()
In handler of main: Class E exception
Resume execution in main
```

In function `f()`, the run time could not find a handler to handle the exception of type `E` thrown. The run time finds a matching handler in a dynamically surrounding try block: the try block in the `main()` function.

If the run time cannot find a matching handler in the program, it calls the **`terminate()`** function.

RELATED REFERENCES

- “The try Keyword” on page 401
- “`terminate()`” on page 416

The throw Expression

C++ You use a *throw expression* to indicate when your program encounters an exception.

Syntax — throw Expression

```
→ throw [assignment_expression] →
```

The type of *assignment_expression* cannot be an incomplete type, or a pointer to an incomplete type other than one of the following:

- **`void*`**
- **`const void*`**
- **`volatile void*`**
- **`const volatile void*`**

The *assignment_expression* is treated the same way as a function argument in a call or the operand of a return statement.

If the *assignment_expression* is a class object, that object's copy constructor and destructor must be accessible. For example, you cannot throw a class object that has its copy constructor declared as private.

RELATED REFERENCES

- “Incomplete Types” on page 71

Rethrowing an Exception

C++ If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression (**throw** without *assignment_expression*) causes the originally thrown object to be rethrown.

Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next dynamically enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred. Any catch blocks for the dynamically enclosing try block have an opportunity to catch the exception.

The following example demonstrates rethrowing an exception:

```
#include <iostream>
using namespace std;

struct E {
    const char* message;
    E() : message("Class E") { }
};

struct E1 : E {
    const char* message;
    E1() : message("Class E1") { }
};

struct E2 : E {
    const char* message;
    E2() : message("Class E2") { }
};

void f() {
    try {
        cout << "In try block of f()" << endl;
        cout << "Throwing exception of type E1" << endl;
        E1 myException;
        throw myException;
    }
    catch (E2& e) {
        cout << "In handler of f(), catch (E2& e)" << endl;
        cout << "Exception: " << e.message << endl;
        throw;
    }
    catch (E1& e) {
        cout << "In handler of f(), catch (E1& e)" << endl;
        cout << "Exception: " << e.message << endl;
        throw;
    }
    catch (E& e) {
        cout << "In handler of f(), catch (E& e)" << endl;
        cout << "Exception: " << e.message << endl;
        throw;
    }
}

int main() {
    try {
        cout << "In try block of main()" << endl;
        f();
    }
}
```

```

    catch (E2& e) {
        cout << "In handler of main(), catch (E2& e)" << endl;
        cout << "Exception: " << e.message << endl;
    }
    catch (...) {
        cout << "In handler of main(), catch (...)" << endl;
    }
}

```

The following is the output of the above example:

```

In try block of main()
In try block of f()
Throwing exception of type E1
In handler of f(), catch (E1& e)
Exception: Class E1
In handler of main(), catch (...)

```

The try block in the main() function calls function f(). The try block in function f() throws an object of type E1 named myException. The handler catch (E1 &e) catches myException. The handler then rethrows myException with the statement throw to the next dynamically enclosing try block: the try block in the main() function. The handler catch (...) catches myException.

RELATED REFERENCES

- “The throw Expression” on page 409

Stack Unwinding

C++ When an exception is thrown and control passes from a try block to a handler, the C++ run time calls destructors for all automatic objects constructed since the beginning of the try block. This process is called *stack unwinding*. The automatic objects are destroyed in reverse order of their construction. (Automatic objects are local objects that have been declared **auto** or **register**, or not declared **static** or **extern**. An automatic object x is deleted whenever the program exits the block in which x is declared.)

If an exception is thrown during construction of an object consisting of subobjects or array elements, destructors are only called for those subobjects or array elements successfully constructed before the exception was thrown. A destructor for a local static object will only be called if the object was successfully constructed.

If during stack unwinding a destructor throws an exception and that exception is not handled, the **terminate()** function is called. The following example demonstrates this:

```

#include <iostream>
using namespace std;

struct E {
    const char* message;
    E(const char* arg) : message(arg) { }
};

void my_terminate() {
    cout << "Call to my_terminate" << endl;
};

struct A {
    A() { cout << "In constructor of A" << endl; }
    ~A() {

```

Stack Unwinding

```
        cout << "In destructor of A" << endl;
        throw E("Exception thrown in A()");
    }
};

struct B {
    B() { cout << "In constructor of B" << endl; }
    ~B() { cout << "In destructor of B" << endl; }
};

int main() {
    set_terminate(my_terminate);

    try {
        cout << "In try block" << endl;
        A a;
        B b;
        throw("Exception thrown in try block of main()");
    }
    catch (const char* e) {
        cout << "Exception: " << e << endl;
    }
    catch (...) {
        cout << "Some exception caught in main()" << endl;
    }

    cout << "Resume execution of main()" << endl;
}
```

The following is the output of the above example:

```
In try block
In constructor of A
In constructor of B
In destructor of B
In destructor of A
Call to my_terminate
```

In the try block, two automatic objects are created: a and b. The try block throws an exception of type **const char***. The handler `catch (const char* e)` catches this exception. The C++ run time unwinds the stack, calling the destructors for a and b in reverse order of their construction. The destructor for a throws an exception. Since there is no handler in the program that can handle this exception, the C++ run time calls **terminate()**. (The function **terminate()** calls the function specified as the argument to **set_terminate()**. In this example, **terminate()** has been specified to call `my_terminate()`.)

RELATED REFERENCES

- “`terminate()`” on page 416
- “`set_unexpected()` and `set_terminate()`” on page 418

Exception Specifications

C++ C++ provides a mechanism to ensure that a given function is limited to throwing only a specified list of exceptions. An exception specification at the beginning of any function acts as a guarantee to the function’s caller that the function will throw only the exceptions contained in the exception specification.

For example, a function:

```
void translate() throw(unknown_word,bad_grammar) { /* ... */ }
```

explicitly states that it will only throw exception objects whose types are `unknown_word` or `bad_grammar`, or any type derived from `unknown_word` or `bad_grammar`.

Syntax – Exception Specification



The *type_id_list* is a comma-separated list of types. In this list you cannot specify an incomplete type, a pointer or a reference to an incomplete type, other than a pointer to **void**, optionally qualified with **const** and/or **volatile**. You cannot define a type in an exception specification.

A function with no exception specification allows all exceptions. A function with an exception specification that has an empty *type_id_list*, **throw()**, does not allow any exceptions to be thrown.

An exception specification is not part of a function's type.

An exception specification may only appear at the end of a function declarator of a function, pointer to function, reference to function, pointer to member function declaration, or pointer to member function definition. An exception specification cannot appear in a typedef declaration. The following declarations demonstrate this:

```

void f() throw(int);
void (*g)() throw(int);
void h(void i() throw(int));
// typedef int (*j)() throw(int);
  
```

The compiler would not allow the last declaration, `typedef int (*j)() throw(int)`.

Suppose that class A is one of the types in the *type_id_list* of an exception specification of a function. That function may throw exception objects of class A, or any class publicly derived from class A. The following example demonstrates this:

```

class A { };
class B : public A { };
class C { };

void f(int i) throw (A) {
    switch (i) {
        case 0: throw A();
        case 1: throw B();
        default: throw C();
    }
}

void g(int i) throw (A*) {
    A* a = new A();
    B* b = new B();
    C* c = new C();
    switch (i) {
        case 0: throw a;
        case 1: throw b;
        default: throw c;
    }
}
  
```

Function `f()` can throw objects of types A or B. If the function tries to throw an object of type C, the compiler will call **unexpected()** because type C has not been

Exception Specifications

specified in the function's exception specification, nor does it derive publicly from A. Similarly, function g() cannot throw pointers to objects of type C; the function may throw pointers of type A or pointers of objects that derive publicly from A.

A function that overloads a virtual function can only throw exceptions specified by the virtual function. The following example demonstrates this:

```
class A {
    public:
        virtual void f() throw (int, char);
};

class B : public A {
    public: void f() throw (int) { }
};

/*
    class C : public A {
        public: void f() { }
    };

    class D : public A {
        public: void f() throw (int, char, double) { }
    };
*/
```

The compiler allows B::f() because the member function may throw only exceptions of type **int**. The compiler would not allow C::f() because the member function may throw any kind of exception. The compiler would not allow D::f() because the member function can throw more types of exceptions (**int**, **char**, and **double**) than A::f().

Suppose that you assign or initialize a pointer to function named x with a function or pointer to function named y. The pointer to function x can only throw exceptions specified by the exception specifications of y. The following example demonstrates this:

```
void (*f)();
void (*g)();
void (*h)() throw (int);

void i() {
    f = h;
    // h = g;
}
```

The compiler allows the assignment f = h because f can throw any kind of exception. The compiler would not allow the assignment h = g because h can only throw objects of type **int**, while g can throw any kind of exception.

Implicitly declared special functions (default constructors, copy constructors, destructors, and copy assignment operators) have exception specifications. An implicitly declared special function will have in its exception specification the types declared in the functions' exception specifications that the special function invokes. If any function that a special function invokes allows all exceptions, then that special function allows all exceptions. If all the functions that a special function invokes allow no exceptions, then that special function will allow no exceptions. The following example demonstrates this:

```
class A {
    public:
        A() throw (int);
        A(const A&) throw (float);
};
```



```

    ~A() throw();
};

class B : public A {
public:
    B() throw (char);
    B(const A&);
    ~B() throw();
};

class C : public B { };

```

The following special functions in the above example have been implicitly declared:

```

C::C() throw (int, char);
C::C(const C&);
C::~~C() throw();

```

The default constructor of C can throw exceptions of type **int** or **char**. The copy constructor of C can throw any kind of exception. The destructor of C cannot throw any exceptions.

RELATED REFERENCES

- “Incomplete Types” on page 71
- “Function Declarations” on page 154
- “Pointers to Functions” on page 173
- “Chapter 15. Special Member Functions” on page 341
- “unexpected()”

Special Exception Handling Functions

> C++ Not all thrown errors can be caught and successfully dealt with by a catch block. In some situations, the best way to handle an exception is to terminate the program. Two special library functions are implemented in C++ to process exceptions not properly handled by catch blocks or exceptions thrown outside of a valid try block. These functions are **unexpected()** and **terminate()**.

RELATED REFERENCES

- “unexpected()”
- “terminate()” on page 416

unexpected()

> C++ When a function with an exception specification throws an exception that is not listed in its exception specification, the C++ run time does the following:

1. The **unexpected()** function is called.
2. The **unexpected()** function calls the function pointed to by **unexpected_handler**. By default, **unexpected_handler** points to the function **terminate()**.

You can replace the default value of **unexpected_handler** with the function **set_unexpected()**.

Although **unexpected()** cannot return, it may throw (or rethrow) an exception. Suppose the exception specification of a function **f()** has been violated. If

Special Exception Handling Functions

unexpected() throws an exception allowed by the exception specification of `f()`, then the C++ run time will search for another handler at the call of `f()`. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct E {
    const char* message;
    E(const char* arg) : message(arg) { }
};

void my_unexpected() {
    cout << "Call to my_unexpected" << endl;
    throw E("Exception thrown from my_unexpected");
}

void f() throw(E) {
    cout << "In function f(), throw const char* object" << endl;
    throw("Exception, type const char*, thrown from f()");
}

int main() {
    set_unexpected(my_unexpected);
    try {
        f();
    }
    catch (E& e) {
        cout << "Exception in main(): " << e.message << endl;
    }
}
```

The following is the output of the above example:

```
In function f(), throw const char* object
Call to my_unexpected
Exception in main(): Exception thrown from my_unexpected
```

The **main()** function's try block calls function `f()`. Function `f()` throws an object of type **const char***. However the exception specification of `f()` allows only objects of type `E` to be thrown. The function **unexpected()** is called. The function **unexpected()** calls `my_unexpected()`. The function `my_unexpected()` throws an object of type `E`. Since **unexpected()** throws an object allowed by the exception specification of `f()`, the handler in the **main()** function may handle the exception.

If **unexpected()** did not throw (or rethrow) an object allowed by the exception specification of `f()`, then the C++ run time does one of two things:

- If the exception specification of `f()` included the class **std::bad_exception**, **unexpected()** will throw an object of type **std::bad_exception**, and the C++ run time will search for another handler at the call of `f()`.
- If the exception specification of `f()` did not include the class **std::bad_exception**, the function **terminate()** is called.

RELATED REFERENCES

- "set_unexpected() and set_terminate()" on page 418

terminate()

C++ In some cases, the exception handling mechanism fails and a call to **void terminate()** is made. This **terminate()** call occurs in any of the following situations:

- The exception handling mechanism cannot find a handler for a thrown exception. The following are more specific cases of this:

Special Exception Handling Functions

- During stack unwinding, a destructor throws an exception and that exception is not handled.
- The expression that is thrown also throws an exception, and that exception is not handled.
- The constructor or destructor of a non-local static object throws an exception, and the exception is not handled.
- A function registered with **atexit()** throws an exception, and the exception is not handled. The following demonstrates this:

```
extern "C" printf(char* ...);
#include <exception>
#include <cstdlib>
using namespace std;

void f() {
    printf("Function f()\n");
    throw "Exception thrown from f()";
}

void g() { printf("Function g()\n"); }
void h() { printf("Function h()\n"); }

void my_terminate() {
    printf("Call to my_terminate\n");
    abort();
}

int main() {
    set_terminate(my_terminate);
    atexit(f);
    atexit(g);
    atexit(h);
    printf("In main\n");
}
```

The following is the output of the above example:

```
In main
Function h()
Function g()
Function f()
Call to my_terminate
```

To register a function with **atexit()**, you pass a parameter to **atexit()** a pointer to the function you want to register. At normal program termination, **atexit()** calls the functions you have registered with no arguments in reverse order.

The **atexit()** function is in the **<cstdlib>** library.

- A throw expression without an operand tries to rethrow an exception, and no exception is presently being handled.
- A function **f()** throws an exception that violates its exception specification. The **unexpected()** function then throws an exception which violates the exception specification of **f()**, and the exception specification of **f()** did not include the class **std::bad_exception**.
- The default value of **unexpected_handler** is called.

The **terminate()** function calls the function pointed to by **terminate_handler**. By default, **terminate_handler** points to the function **abort()**, which exits from the program. You can replace the default value of **terminate_handler** with the function **set_terminate()**.

A terminate function cannot return to its caller, either by using **return** or by throwing an exception.

Special Exception Handling Functions

RELATED REFERENCES

- “set_unexpected() and set_terminate()”

set_unexpected() and set_terminate()

C++ The function **unexpected()**, when invoked, calls the function most recently supplied as an argument to **set_unexpected()**. If **set_unexpected()** has not yet been called, **unexpected()** calls **terminate()**.

The function **terminate()**, when invoked, calls the function most recently supplied as an argument to **set_terminate()**. If **set_terminate()** has not yet been called, **terminate()** calls **abort()**, which ends the program.

You can use **set_unexpected()** and **set_terminate()** to register functions you define to be called by **unexpected()** and **terminate()**. The functions **set_unexpected()** and **set_terminate()** are included in the standard header files. Each of these functions has as its return type and its argument type a pointer to function with a **void** return type and no arguments. The pointer to function you supply as the argument becomes the function called by the corresponding special function: the argument to **set_unexpected()** becomes the function called by **unexpected()**, and the argument to **set_terminate()** becomes the function called by **terminate()**.

Both **set_unexpected()** and **set_terminate()** return a pointer to the function that was previously called by their respective special functions (**unexpected()** and **terminate()**). By saving the return values, you can restore the original special functions later so that **unexpected()** and **terminate()** will once again call **terminate()** and **abort()**.

If you use **set_terminate()** to register your own function, the function should not return to its caller but terminate execution of the program.

VAC++ If you attempt to return from the function called by **terminate()**, **abort()** is called instead and the program ends.

Example of Using the Exception Handling Functions

C++ The following example shows the flow of control and special functions used in exception handling:

```
#include <iostream>
#include <exception>
using namespace std;

class X { };
class Y { };
class A { };

// pfv type is pointer to function returning void
typedef void (*pfv)();

void my_terminate() {
    cout << "Call to my terminate" << endl;
    abort();
}

void my_unexpected() {
    cout << "Call to my_unexpected()" << endl;
    throw;
}
```

```

void f() throw(X,Y, bad_exception) {
    throw A();
}

void g() throw(X,Y) {
    throw A();
}

int main()
{
    pfv old_term = set_terminate(my_terminate);
    pfv old_unex = set_unexpected(my_unexpected);
    try {
        cout << "In first try block" << endl;
        f();
    }
    catch(X) {
        cout << "Caught X" << endl;
    }
    catch(Y) {
        cout << "Caught Y" << endl;
    }
    catch (bad_exception& e1) {
        cout << "Caught bad_exception" << endl;
    }
    catch (...) {
        cout << "Caught some exception" << endl;
    }

    cout << endl;

    try {
        cout << "In second try block" << endl;
        g();
    }
    catch(X) {
        cout << "Caught X" << endl;
    }
    catch(Y) {
        cout << "Caught Y" << endl;
    }
    catch (bad_exception& e2) {
        cout << "Caught bad_exception" << endl;
    }
    catch (...) {
        cout << "Caught some exception" << endl;
    }
}

```

The following is the output of the above example:

```

In first try block
Call to my_unexpected()
Caught bad_exception

```

```

In second try block
Call to my_unexpected()
Call to my_terminate

```

At run time, this program behaves as follows:

1. The call to **set_terminate()** assigns to `old_term` the address of the function last passed to **set_terminate()** when **set_terminate()** was previously called.

Special Exception Handling Functions

2. The call to **set_unexpected()** assigns to `old_unex` the address of the function last passed to **set_unexpected()** when **set_unexpected()** was previously called.
3. Within the first try block, function `f()` is called. Because `f()` throws an unexpected exception, a call to **unexpected()** is made. **unexpected()** in turn calls `my_unexpected()`, which prints a message to standard output. The function `my_unexpected()` tries to rethrow the exception of type A. Because class A has not been specified in the exception specification of function `f()`, `my_unexpected()` throws an exception of type **bad_exception**.
4. Because **bad_exception** has been specified in the exception specification of function `f()`, the handler `catch (bad_exception& e1)` is able to handle the exception.
5. Within the second try block, function `g()` is called. Because `g()` throws an unexpected exception, a call to **unexpected()** is made. The **unexpected()** throws an exception of type **bad_exception**. Because **bad_exception** has not been specified in the exception specification of `g()`, **unexpected()** calls **terminate()**, which calls the function `my_terminate()`.
6. `my_terminate()` displays a message then calls **abort()**, which terminates the program.

Note that the catch blocks following the second try block are not entered, because the exception was handled by `my_unexpected()` as an unexpected throw, not as a valid exception.

RELATED REFERENCES

- “`unexpected()`” on page 415
- “`terminate()`” on page 416
- “`set_unexpected()` and `set_terminate()`” on page 418

Appendix A. C and C++ Compatibility on the z/OS Platform

z/OS This appendix pertains to the differences between C and C++ that apply specifically to the z/OS platform. The contents describe the constructs that are found in both ISO C and ISO C++, but which are treated differently in the two languages, and interactions with other products that do not support C++.

RELATED REFERENCES

- “Constructs Found in Both C++ and C”
- “Interactions with Other Products” on page 424

Constructs Found in Both C++ and C

Because ISO C++ is based on ISO C, the two languages have many constructs in common. The use of some of these shared constructs differs as described in this section.

Character Array Initialization

In C++, when you initialize character arrays, a trailing `'\0'` (zero of type `char`) is appended to the string initializer. You cannot initialize a character array with more initializers than there are array elements.

In C, space for the trailing `'\0'` can be omitted in this type of initialization.

The following initialization, for instance, is not valid in C++:

```
char v[3] = "asd"; // not valid in C++, valid in C
```

because four elements are required. This initialization produces an error because there is no space for the implied trailing `'\0'` (zero of type `char`).

Class and typedef Names

In C++, a class and a typedef cannot both use the same name to refer to a different type within the same scope (unless the typedef is a synonym for the class name). In C, a typedef name and a struct tag name declared in the same scope can have the same name because they have different name spaces. For example:

```
int main ()
{
    typedef double db;
    struct db; // error in C++, valid in C

    typedef struct st st; // valid C and C++
}
```

Class and Scope Declarations

In C++, a class declaration introduces the class name into the scope where it is declared and hides any object, function, or other declaration of that name in an enclosing scope. In C, an inner scope declaration of a struct name does not hide an object or function of that name in an outer scope. For example:

```
double db;
int main ()
{
    struct db // hides double object db in C++
}
```

Constructs Found in Both C++ and C

```
{ char* str; };  
int x = sizeof(db);    // size of struct in C++  
                        // size of double in C  
}
```

const Object Initialization

In C++, const objects must be initialized. In C, they can be left uninitialized.

Definitions

An object declaration is a definition in C++. In C, it is a declaration (formerly known as a tentative definition). For example:

```
int i;
```

In C++, a global data object must be defined only once. In C, a global data object can be declared several times without using the extern keyword.

In C++, multiple definitions for a single variable cause an error. A C compilation unit can contain many identical declarations for a variable.

Definitions within Return or Argument Types

In C++, types may not be defined in return or argument types. C allows such definitions. For example, the following declarations produce errors in C++, but are valid declarations in C:

```
void print(struct X { int i;} x);    // error in C++  
enum count{one, two, three} counter(); // error in C++
```

Enumerator Type

An enumerator has the same type as its enumeration in C++. In C, an enumeration has type int.

Enumeration Type

The assignment to an object of enumeration type with a value that is not of that enumeration type produces an error in C++. In C, an object of enumeration type can be assigned values of any integral type.

Function Declarations

In C++, all declarations of a function must match the unique definition of a function. C has no such restriction.

Functions with an Empty Argument List

Consider the following function declaration:

```
int f();
```

In C++, this function declaration means that the function takes no arguments. In C, it could take any number of arguments, of any type.

Global Constant Linkage

In C++, an object declared const has internal linkage, unless it has previously been given external linkage. In C, it has external linkage.

Jump Statements

C++ does not allow you to jump over declarations containing initializations. C does allow you to use jump statements for this purpose.

Keywords

C++ contains some additional keywords not found in C. C programs that use these keywords as identifiers are not valid C++ programs:

Table 9. C++ Keywords

asm	export	private	true
bool	false	protected	try
catch	friend	public	typeid
class	inline	reinterpret_cast	typename
const_cast	mutable	static_cast	using
delete	namespace	template	virtual
dynamic_cast	new	this	wchar_t
explicit	operator	throw	

main() Recursion

In C++, `main()` cannot be called recursively and cannot have its address taken. C allows recursive calls and allows pointers to hold the address of `main()`.

Names of Nested Classes

In C++, the name of a nested class is local to its enclosing class. In C, the name of the nested structure belongs to the same scope as the name of the outermost enclosing structure.

Pointers to void

C++ allows void pointers to be assigned only to other void pointers. In C, a pointer to void can be assigned to a pointer of any other type without an explicit cast.

Prototype Declarations

C++ requires full prototype declarations. C allows nonprototyped functions.

Return without Declared Value

In both C and C++, `main()` must be declared to return a value of type `int`. In C++, if no value is explicitly returned from `main()` by means of a return statement and if program execution reaches the end of function `main` (that is, the program does not terminate due to a call to `exit()`, `std::terminate()`, or a similar function), then the value 0 is implicitly returned. A return (either explicit or implicit) from all other functions that are declared to return a value *must* return a value. In C, a function that is declared to return a value can return with no value, with unspecified results.

__STDC__ Macro

The predefined macro variable `__STDC__` is not defined for C++. It has the integer value 0 when it is used a `#if` statement, indicating that the C++ language is not a proper superset of C, and that the compiler does not conform to C. In C, `__STDC__` has the integer value 1.

typedefs in Class Declarations

In C++, a typedef name may not be redefined in a class declaration after being used in the declaration. C allows such a declaration. For example:

Constructs Found in Both C++ and C

```
int main ()
{
    typedef double db;
    struct st
    {
        db x;
        double db; // error in C++, valid in C
    };
}
```

Interactions with Other Products

You cannot write a C++ program that includes interfaces to Cross-System Product (CSP). However, you can write a C program to access CSP and call the C program from a C++ program.

In general, application libraries that provide C interfaces may not support applications written in C++ if their header files do not conform to C++ syntax.

Appendix B. Common Usage C Language Level for the z/OS Platform

z/OS The X/Open Portability Guide (XPG) Issue 3 describes a C language definition referred to as Common Usage C. This language definition is roughly equivalent to K&R C, and differs from the ISO C language definition. It is based on various C implementations that predate the ISO standard.

Common Usage C is supported with the `LANGLVL(COMMONC)` compiler option or the `#pragma langlvl(commonc)` directive. These cause the compiler to accept C source code containing Common Usage C constructs.

Many of the Common Usage C constructs are already supported by `#pragma langlvl(extended)`. The following language elements are different from those accepted by `#pragma langlvl(extended)`.

- Standard integral promotions preserve sign. For example, unsigned char or unsigned short are promoted to unsigned int. This is functionally equivalent to specifying the `UPCONV` compiler option.
- Trigraphs are not processed in string or character literals. For example, consider the following source line:

```
??=define STR "??= not processed"
```

The above line gets preprocessed to:

```
#define STR "??= not processed"
```

- The `sizeof` operator is permitted on bitfields. The result is the size of an unsigned int (4).
- Bitfields other than type `int` are permitted. The compiler issues a warning and changes the type to unsigned int.
- Macro parameters found within single or double quotation marks are expanded. For example, consider the following source lines:

```
#define STR(AAA) "String is: AAA"  
#define ST STR(BBB)
```

The above lines are preprocessed to:

```
#define STR(AAA) "String is: AAA"  
#define ST "String is: BBB"
```

- Macros can be redefined without first being undefined (that is, without an intervening `#undef`). An informational message is issued saying that the second definition is used.
- The empty comment (`/**/`) in a function-like macro is equivalent to the ISO token concatenation operator `##`.

The `LANGLVL` compiler option is described in the *z/OS C/C++ User's Guide*. The `#pragma langlvl` is described in "langlvl" on page 237.

Appendix C. Conforming to POSIX 1003.1

z/OS The implementation resulting from the combination of z/OS UNIX and the z/OS Language Environment supports the ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990 standard. POSIX stands for Portable Operating System Interface.

See the *OpenEdition POSIX.1 Conformance Document for POSIX on MVS/ESA: IEEE Standard 1003.1-1990*, GC23-3011, for a description of how the z/OS UNIX implementation meets the criteria.

Appendix D. Supporting ISO Standards

z/OS This appendix describes changes made to the z/OS C/C++ Compiler and Library to support the *American National Standard for Information Systems - Programming Language C* standard. It also describes implementation-defined behavior of the z/OS C/C++ compiler that is not defined by ISO.

The following sections describe how the z/OS C compiler defines some of the implementation-specific behavior from the ISO C Standard. In-depth usage information is provided in *z/OS C/C++ User's Guide* and *z/OS C/C++ Programming Guide*.

- "Identifiers"
- "Characters" on page 430
- "String Conversion" on page 430
- "Integers" on page 431
- "Floating-Point" on page 431
- "Arrays and Pointers" on page 432
- "Registers" on page 432
- "Structures, Unions, Enumerations, Bit Fields" on page 432
- "Declarators" on page 433
- "Statements" on page 433
- "Preprocessing Directives" on page 433
- "Library Functions" on page 433
- "Error Handling" on page 434
- "Signals" on page 435
- "Translation Limits" on page 435
- "Streams, Records, and Files" on page 436
- "Memory Management" on page 437
- "Environment" on page 437
- "Localization" on page 438
- "Time" on page 438

Identifiers

The number of significant characters in an identifier with no external linkage:

- 1024

The number of significant characters in an identifier with external linkage:

- 1024 with the compile-time option `LONGNAME` specified
- 8 otherwise

The C++ compiler truncates external identifiers without C++ linkage after 8 characters if the `NOLONGNAME` compiler option or `#pragma` is in effect.

Case sensitivity of external identifiers:

- The linkage editor accepts all external names up to 8 characters, and may not be case sensitive. The binder accepts all external names up to 1024 characters, and is optionally case sensitive. The linkage editor accepts all external names up to 8 characters, and may not be case sensitive, depending on whether you use the `NOLONGNAME` compiler option or `#pragma`. When using the z/OS C compiler with the `NOLONGNAME` option, all external names are truncated to 8 characters. As an aid to portability, identifiers that differ only in case after truncation are flagged as an error.

Characters

Source and execution characters which are not specified by the ISO standard:

- The caret (^) character in ASCII (bitwise exclusive OR symbol) or the equivalent not (~) character in EBCDIC.
- The vertical broken line (!) character in ASCII which may be represented by the vertical line (|) character on EBCDIC systems.

Shift states used for the encoding of multibyte characters:

- The shift states are indicated with the SHIFTOUT (hex value \x0E) characters and SHIFTIN (hex value \x0F).

The number of bits that represent a character:

- 8 bits

The mapping of members of the source character set (characters and strings) to the execution character set:

- The same code page is used for the source and execution character set.

The value of an integer character constant that contains a character/escape sequence not represented in the basic execution character set:

- A warning is issued for an unknown character/escape sequence and the char is assigned the character following the back slash.

The value of a wide character constant that contains a character/escape sequence not represented in the extended execution character set:

- A warning is issued for the unknown character/escape sequence and the wchar_t is assigned the wide character following the back slash.

The value of an integer character constant that contains more than one character:

- The lowest four bytes represent the character constant.

The value of a wide character constant that contains more than one multibyte character:

- The lowest four bytes of the multibyte characters are converted to represent the wide character constant.

Equivalent type of char: signed char, unsigned char, or user-defined:

- The default for char is unsigned

Is each sequence of white-space characters (excluding the new-line) retained or replaced by one space character?

- Any spaces or comments in your source program will be interpreted as one space.

String Conversion

Additional implementation-defined sequence forms that can be accepted by strtod(), strtol() and strtoul() functions in other than the C locale:

- None

Integers

Table 10. Integers

Type	Amount of Storage	Range (in limits.h)
signed short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
signed int	4 bytes	-2,147,483,647 minus 1 to 2,147,483,647
unsigned int	4 bytes	0 to 4,294,967,295
signed long	4 bytes	-2,147,483,647 minus 1 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295
signed long long	8 bytes	-9,223,372,036,854,775,807 minus 1 to 9,223,372,036,854,775,807
unsigned long long	8 bytes	0 to 18,446,744,073,709,551,615

The result of converting an integer to a signed char :

- The lowest 1 byte of the integer is used to represent the char.

The result of converting an integer to a shorter signed integer:

- The lowest 2 bytes of the integer are used to represent the short int.

The result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented:

- The bit pattern is preserved and the sign bit has no significance.

The result of bitwise operations (|, &, ^) on signed int:

- The representation is treated as a bit pattern and 2's complement arithmetic is performed.

The sign of the remainder of integer division if either operand is negative:

- The remainder is negative if exactly one operand is negative.

The result of a right shift of a negative-valued signed integral type:

- The result is sign extended and the sign is propagated.

Floating-Point

Table 11. Floating Point

Type	Amount of Storage	Range (approximate)	
		IBM S/390 Hexadecimal Format	IEEE Binary Format
float	4 bytes	5.5×10^{-79} to 7.2×10^{75}	1.2×10^{-38} to 3.4×10^{38}
double	8 bytes	5.5×10^{-79} to 7.2×10^{75}	2.2×10^{-308} to 1.8×10^{308}
long double	16 bytes	5.5×10^{-79} to 7.2×10^{75}	3.4×10^{-4932} to 1.2×10^{4932}

The following is the direction of truncation (or rounding) when you convert an integer number to an IBM S/390 hexadecimal floating-point number, or to an IEEE binary floating-point number:

- IBM S/390 hexadecimal format:

ISO standards support

When the floating-point cannot exactly represent the original value, the value is truncated.

When a floating-point number is converted to a narrower floating-point number, the floating-point number is truncated.

- IEEE binary format:

The rounding direction is determined by the `ROUND` compiler option. The `ROUND` option only affects the rounding of floating-point values that the z/OS C/C++ compiler can evaluate at compile time. It has no effect on rounding at run time.

Arrays and Pointers

The type of `size_t`:

- `unsigned int`

The type of `ptrdiff_t`:

- `int`

The result of casting a pointer to an integer:

- The bit patterns are preserved.

The result of casting an integer to a pointer:

- The bit patterns are preserved.

Registers

The effect of the `register` storage class specifier on the storage of objects in registers:

- The register storage class indicates to the compiler that a variable in a block scope data definition or a parameter declaration is heavily used (such as a loop control variable). It is equivalent to `auto`, except that the compiler might, if possible, place the variable into a machine register for faster access.

Structures, Unions, Enumerations, Bit Fields

The result when a member of a union object is accessed using a member of a different type:

- The result is undefined.

The alignment/padding of structure members:

- If the structure is not packed, then padding is added to align the structure members on their natural boundaries. If the structure is packed, no padding is added.

The padding at the end of structure/union:

- Padding is added to end the structure on its natural boundary. The alignment of the `struct` or union is that of its strictest member.

The type of an `int` bit field (signed `int`, unsigned `int`, user defined):

- The default is unsigned.

The order of allocation of bit fields within an `int` :

- Bit fields are allocated from low memory to high memory. For example, `0x12345678` would be stored with byte 0 containing `0x12`, and byte 3 containing `0x78`.

The rule for bit fields crossing a storage unit boundary:

- Bit fields can cross storage unit boundaries.

The integral type that represents the values of an enumeration type:

- Enumerations can have the type `char`, `short`, or `long` and be either signed or unsigned depending on their smallest and largest values.

Declarators

The maximum number of declarators (pointer, array, function) that can modify an arithmetic, structure, or union type:

- The only constraint is the availability of system resources.

Statements

The maximum number of case values in a switch statement:

- Because the case values must be integers and cannot be duplicated, the limit is `INT_MAX`.

Preprocessing Directives

Does the value of a single-character constant in a constant expression that controls conditional inclusion match the value of the character constant in the execution character set?

- Yes

Can such a constant have a negative value?

- Yes

The method of searching include source files (`< >`):

- See *z/OS C/C++ User's Guide*.

Is the search for quoted source file names supported ("`...`")?

- User include files can be specified in double quotes. See *z/OS C/C++ User's Guide*.

The mapping between the name specified in the include directive and the external source file name:

- See *z/OS C/C++ User's Guide*.

The behavior of each pragma directive:

- See "Pragma Directives (`#pragma`)" on page 219.

The definitions of `__DATE__` and `__TIME__` when date and time of translation is not available:

- For *z/OS C/C++*, the date and time of translation are always available.

Library Functions

The definition of `NULL` macro:

- `NULL` is defined to be a `((void *)0)`.

The format of diagnostic printed by the `assert` macro, and the termination behavior (abort behavior):

- When `assert` is executed, if the expression is false, the diagnostic message written by the `assert` macro has the format:

ISO standards support

Assertion failed: *expression*, file *filename*, line *line-number*

Set of characters tested by the `isxxxx` functions:

- To create a table of the characters set up by the `ctype` functions use the program in the following example.

CCNRABG

```
/* this example prints out ctest characters */

#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int ch;

    for (ch = 0; ch <= 0xff; ch++)
    {
        printf("%#04X ", ch);
        printf("%3s ", isalnum(ch) ? "AN" : " ");
        printf("%2s ", isalpha(ch) ? "A" : " ");
        printf("%2s", iscntrl(ch) ? "C" : " ");
        printf("%2s", isdigit(ch) ? "D" : " ");
        printf("%2s", isgraph(ch) ? "G" : " ");
        printf("%2s", islower(ch) ? "L" : " ");
        printf("%c", isprint(ch) ? ch : ' ');
        printf("%3s", ispunct(ch) ? "PU" : " ");
        printf("%2s", isspace(ch) ? "S" : " ");
        printf("%3s", isprint(ch) ? "PR" : " ");
        printf("%2s", isupper(ch) ? "U" : " ");
        printf("%2s", isxdigit(ch) ? "X" : " ");

        putchar('\n');
    }
}
```

The result of calling `fmod()` function with the second argument zero (return zero, domain error):

- `fmod()` returns a 0.

Error Handling

The format of the message generated by the `perror()` and `strerror()` functions:

- See *z/OS Language Environment Run-Time Messages* for the messages emitted for `perror()` and `strerror()`.

Note: `errno` is not emitted with the message.

How diagnostic messages are recognized:

- Error handling is both a compile-time and run-time behavior. Refer to *z/OS C/C++ Messages Guide* and *z/OS Language Environment Run-Time Messages* for the lists of z/OS C/C++ messages provided.

The different classes of messages:

- The message summary of a listing uses message types, return codes, and numeric severity levels, which are shown in the table below.

Return Code	Message Type—C++	Message Type—C	Numeric Severity Level—C
0	Informational (I)	Informational	00

Return Code	Message Type—C++	Message Type—C	Numeric Severity Level—C
4	Warning (W)	Warning	10
8	Error (E)	n/a	n/a
12	Severe (S)	Error	30
16	Unrecoverable (U)	Severe Error	>30

How the level of diagnostics can be controlled:

- Use the compile-time option `FLAG` to control the level of diagnostics. The C compiler option `CHECKOUT` and the C++ compiler option `INFO` provide programming style diagnostics to aid you in determining possible programming errors.

Signals

The set of signals for the `signal()` function:

- See *z/OS C/C++ Programming Guide*.

The parameters and the usage of each signal recognized by the `signal()` function:

- See *z/OS C/C++ Programming Guide*.

The default handling and the handling at program start-up for each signal recognized by `signal()` function:

- `SIG_DFL` is the default signal.

The signal blocking performed if the equivalent of `signal(sig, SIG_DFL)` is not executed at the beginning of signal handler:

- See *z/OS C/C++ Programming Guide*.

Is the default handling reset if a `SIGKILL` is received by a signal handler?

- Whenever you enter the signal handler, `SIG_DFL` becomes the default.

Translation Limits

System-determined means that the limit is determined by your system resources.

Table 12. Translation Limits

Nesting levels of:

- | | |
|---|---------------------|
| • Compound statements | • System-determined |
| • Iteration control | • System-determined |
| • Selection control | • System-determined |
| • Conditional inclusion | • System-determined |
| • Parenthesized declarators | • System-determined |
| • Parenthesized expression | • System-determined |
| Number of pointer, array and function declarators modifying an arithmetic a structure, a union, and incomplete type declaration | • System-determined |

Significant initial characters in:

- | | |
|---|--------|
| • Internal identifiers | • 1024 |
| • Macro names | • 1024 |
| • C external identifiers (<i>without</i> <code>LONGNAME</code>) | • 8 |
| • C external identifiers (<i>with</i> <code>LONGNAME</code>) | • 1024 |
| • C++ external identifiers | • 1024 |

ISO standards support

Table 12. Translation Limits (continued)

Number of:

- | | |
|---|---------------------|
| • External identifiers in a translation unit | • System-determined |
| • Identifiers with block scope in one block | • System-determined |
| • Macro identifiers simultaneously declared in a translation unit | • System-determined |
| • Parameters in one function definition | • System-determined |
| • Arguments in a function call | • System-determined |
| • Parameters in a macro definition | • System-determined |
| • Parameters in a macro invocation | • System-determined |
| • Characters in a logical source line | • 32760 under MVS |
| • Characters in a string literal | • 32K minus 1 |
| • Bytes in an object | • LONG_MAX (See 1) |
| • Nested include files | • SHRT_MAX |
| • Enumeration constants in an enumeration | • System-determined |
| • Levels in nested structure or union | • System-determined |

Note:

- 1 LONG_MAX is the limit for automatic variables only. For all other variables, the limit is 16 Megabytes.

Streams, Records, and Files

Does the last line of a text stream require a terminating new-line character?

- No, the last new-line character is defaulted.

Do space characters, that are written out to a text stream immediately before a new-line character, appear when read?

- White-space characters written to fixed record format text streams before a new-line do not appear when read. However, white-space characters written to variable record format text streams before a new-line character appear when read.

The number of null characters that can be appended to the end of the binary stream:

- No limit

Where is the file position indicator of an append-mode stream initially positioned?

- The file position indicator is positioned at the end of the file.

Does a write on a text stream cause the associated file to be truncated?

- Yes

Does a file of zero length exist?

- Yes

The rules for composing a valid file name:

- See *z/OS C/C++ Programming Guide*.

Can the same file be simultaneously opened multiple times?

- For reading, the file can be opened multiple times; for writing/appending, the file can be opened once. Once a file is opened for reading, it cannot be opened for writing.

The effect of the `remove()` function on an open file:

- `remove()` fails.

The effect of the `rename()` function on file to a name that exists prior to the function call:

- The `rename()` fails.

Are temporary files removed if the program terminates abnormally?

- Yes

The effect of calling the `tmpnam()` function more than `TMP_MAX` times:

- `tmpnam()` fails and returns `NULL`.

The output of `%p` conversion in the `fprintf()` function:

- It is equivalent to `%X`.

The input of `%p` conversion in the `fscanf()` function:

- The value is treated as an integer.

The interpretation of a `-` character that is neither the first nor the last in the scanlist for `%[` conversion in the `fscanf()` function:

- The sequence of characters on either side of the `-` are used as delimiters. For example, `%[a-f]` will read in characters between `'a'` and `'f'`.

The value of `errno` on failure of `fgetpos()` and `ftell()` functions:

- This depends on the failure. For a list of the messages associated with `errno`, see *z/OS Language Environment Run-Time Messages*.

Memory Management

The behavior of `calloc()`, `malloc()` and `realloc()` functions if the size requested is zero:

- Nothing is performed for `calloc()` and `malloc()`; `realloc()` frees the storage.

Environment

The arguments of `main` function:

- You can pass arguments to `main` through `argv` and `argc`.

What happens with open files when the `abort()` function is called?

- The files are closed.

What is returned to the host environment when the `abort()` function is called?

- The return code of 2000 is returned.

The form of successful termination when the `exit` function is called with argument zero or `EXIT_SUCCESS`:

- All files are closed, all storage is released and the return code of 0 is returned.

The form of unsuccessful termination when the `exit` function is called with argument `EXIT_FAILURE`:

- All files are closed, all storage is released and the return code of `EXIT_FAILURE` is returned.

What status is returned by the `exit` function if the argument is other than zero, `EXIT_FAILURE` and `EXIT_SUCCESS`?

- The return code 4096 is returned.

The set of environmental names:

ISO standards support

- There are no environmental names.

The method of altering the environment list obtained by a call to the `getenv()` function:

- See how to execute a command in *z/OS C/C++ Run-Time Library Reference*.

The format and a mode of execution of a string on a call to the `system()` function:

- See *z/OS C/C++ Run-Time Library Reference*.

Localization

The environment specified by the "" locale on a `setlocale()` call:

- EDC\$SAAC

The supported locales:

- See *z/OS C/C++ Programming Guide*.

Time

The local time zone and Daylight Saving Time:

- This is specified in the locale.

The era for the `clock()` function:

- The era starts when the program is started by either a call from the operating system, or a call to `system()`. Under TSO, the era starts when you log on to the system. To measure the time spent in a program, call the `clock()` function at the start of the program, and subtract its return value from the value returned by subsequent calls to `clock()`.

Notices

Note to U.S. Government Users Restricted Rights -- use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd.
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2001. All rights reserved.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtains the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

400	Language Environment	RACF
AD/Cycle	MVS/DFP	S/390
AIX	MVS/ESA	RS/6000
AIX/6000	Network Station	SAA
AS/400	Object Connection	SOM
BookManager	Open Class	SOMobjects
C Set ++	OpenEdition	SP
C++/MVS	OS/2	System/370
C/370	OS/390	System/390
CICS	OS/400	System Object Model
CICS/ESA	Operating System/2	Systems Application Architecture
DB2 Universal Database	Operating System/400	TeamConnection
DFSMS/MVS	PowerPC 403	VisualAge
DRDA	PowerPC 601	VM/ESA
GDDM	PowerPC 603	VSE/ESA
Hiperspace	PowerPC 604	WebSphere
IBM	Presentation Manager	Workplace Shell
IMS	QMF	z/OS
IMS/ESA		

Lotus, Lotus Notes, and Domino are trademarks or registered trademarks of the Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Management Environment, TME 10, and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States and/or other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and/or other countries.

ActionMedia, Itanium, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, or other countries, or both.

C-bus is a registered trademark of Corollary, Inc.

PC Direct is a registered trademark of Ziff Communications Company and is used by IBM Corporation under license

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

Industry Standards

The following standards are supported:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National Standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).

Index

Special Characters

[] (array subscript operator) 106
?: (conditional operators) 137
/= (compound assignment operator) 135
*= (compound assignment operator) 135
&= (compound assignment operator) 135
+= (compound assignment operator) 135
-- (decrement operator) 114
== (equal to operator) 129
++ (increment operator) 114
&& (logical AND operator) 132
!= (not equal to operator) 129
% (remainder) 126
:: (scope resolution operator) 102
+ (addition operator) 126
& (address operator) 116
& (bitwise AND operator) 130
, (comma operator) 140
\ continuation character 29, 196
/ (division operator) 126
. (dot operator) 107
\ escape character 13
* (indirection operator) 117
! (logical negation operator) 115
* (multiplication operator) 125
= (simple assignment operator) 134
- (subtraction operator) 127
- (unary minus operator) 115
+ (unary plus operator) 115
__callback keyword 20
__cdecl 20, 75, 173
-> (arrow operator) 107
__packed 20
preprocessor directive character 196
preprocessor operator 200
~ (bitwise negation operator) 116
^= (compound assignment operator) 135
^ (bitwise exclusive OR operator) 130
^ (caret), locale 12
_Export 20, 77
>= (greater than or equal to operator) 128
> (greater than operator) 128
>>= (compound assignment operator) 135
>> (right-shift operator) 127
<= (less than or equal to operator) 128
< (less than operator) 128
<<= (compound assignment operator) 135
<< (left-shift operator) 127
_Packed 20
| (bitwise inclusive OR operator) 131
| (vertical bar), locale 12
|| (logical OR operator) 132

Numerics

370 macro 211

A

aborting functions 417
abstract classes 336, 339
access
 base classes 321
 friends 314
 members 308
 multiple 329
 private 308, 321
 protected 308
 protected members 320
 public 308, 321
 specifiers 308, 321
 virtual function 338
access specifiers 293, 308
accessibility 308, 329
addition operator (+) 126
address operator (&) 116
aggregates 344
ALIAS compile option 241
alignment
 changing with #pragma pack 249
alignment rules
 nested structures 59
 structures 58
 unions 64
allocation expressions 119
allocation functions 123
alternative representations
 operators and punctuators 20
ambiguities
 base classes 328, 329
 resolving 178
 virtual functions 337
AND operator, bitwise (&) 130
AND operator, logical (&&) 132
ANSI flagging 237
ANSI macro 205
ARCH macro 205
argc (argument count) 163
 example 163
 restrictions 166
arguments
 command-line 165
 default 169
 evaluation 171
 main function 163
 of catch blocks 407
 passing 164
 restrictions 165
 passing by reference 167
 passing by value 167
argv (argument vector) 163
 example 163
 restrictions 165
arithmetic conversions 149

- arrays
 - array-to-pointer conversions 147
 - class members 294
 - declaration 86
 - description 86
 - initializing 88
 - ISO support 432
 - subscripting operator 106
- ASCII character codes 13
- asm 20
- associativity of operators 95
- atexit function 416
- auto storage class specifier 35

B

- base classes
 - abstract 339
 - access 321
 - ambiguities 328, 329
 - description 317
 - direct 327
 - indirect 327
 - initialization 346
 - multiple 327
 - multiple access 329
 - pointers to 319, 320
 - virtual 328, 332
- base list 317, 327
- best viable function 280
- BFP macro 205
- binary expressions 124
- binary operators 124
- binding virtual functions 333
- bit fields 55
 - ISO support 432
- bitwise negation operator (~) 116
- block statement 179
- block visibility 2
- BOOL macro 205
- boolean conversions 145
- boolean literals 31
- boolean variables 46
- boundaries, data 80
- brackets [] 106
- break statement 190

C

- candidate functions 269, 280
- case label 182
- cast expressions 135
- catch blocks 401, 403
 - argument matching 407
 - order of catching 408
- CHAR_SIGNED macro 205
- char type specifier 45
- CHAR_UNSIGNED macro 205
- character
 - data types 45
- character literals 28

- chars pragma 224
- CHARSET_LIB macro 205
- checkout pragma 225
- class key 283
- class member access operators
 - description 107
- class member lists 293
- class members 293
 - ambiguities 331
 - initialization 346
- class names
 - scope 287
- class object 34
- class templates
 - declaration and definition 376
 - distinction from *template class* 374
 - explicit specialization 393
 - member functions 377
 - static data members 377
- classes 286
 - abstract 339
 - class templates 374
 - declarations 283
 - derivation 317
 - friends 310
 - incomplete declarations 288
 - inheritance 315
 - local 290
 - member access 308
 - member functions 295
 - member lists 293
 - member scope 297
 - nested 288, 312
 - objects 34
 - overview 283
 - packing
 - using #pragma pack 249
 - scope 287
 - static members 303
 - this pointer 300
 - using declaration 322
 - virtual 333
 - virtual base 328
 - virtual member functions 333
- COBOL linkage 239
- CODESET macro 206
- comma expressions 140
- comma operator 140
- comment pragma 225
- comments 16
- COMMONC macro 206
- compatible types 137
- COMPATMATH macro 206
- compile options
 - overriding defaults 247
 - specifying 247
- COMPILER_VER macro 206
- compound statement 179
- conditional compilation directives 213
 - elif preprocessor directive 215
 - else preprocessor directive 216

conditional compilation directives (*continued*)

 endif preprocessor directive 217

 examples 217

 if preprocessor directive 215

 ifdef preprocessor directive 215

 ifndef preprocessor directive 216

conditional expression (? :) 137

const_cast 110

const qualifier 69

constant

 member functions 296

constant expressions 101

constant initializers 293

constants

 fixed-point decimal 27

constructors 342

 converting 360

 copy 362

 default 343

 exception handling 411

 initialization

 explicit 344

 initializer list 159

 nontrivial 343, 351

 overview 341

 trivial 343, 351

continuation character 29, 196

continue statement 190

conversion

 constructors 360

 function 361

conversions

 arithmetic 149

 array-to-pointer 147

 boolean 145

 cast 135

 derived class 332

 derived-to- base 147

 explicit keyword 150

 floating- point 146

 function arguments 148

 function-to-pointer 147

 integral 145

 lvalue-to- rvalue 145

 pointer 146

 pointer to member 148

 qualification 148

 reference 147

 standard 144

 user-defined 358

 void pointer 147

converting constructor 150, 360

convlit pragma 226

copy assignment operators 363

copy constructors 362

covariant virtual functions 335

CPLUSPLUS macro 205

CPPUNWIND macro 206

csect pragma 227

D

data members

 description 294

 scope 297

 static 304

DATE macro 204

dbx xxv

deallocation expressions 122

deallocation functions 123

debug tool xxii

debugging

 dbx xxv

 debug tool xxii

decimal data type operators 118

decimal integer literals 24

declarations

 class

 description 283

 incomplete 288

 syntax 283

 description 33

 friend 310, 314

 function

 resolving ambiguities 178

 overview 33

 pointers to members 298

 resolving ambiguous statements 178

 unsubscripted arrays 87

declarators

 description 73

 restrictions 433

decrement operator (--) 114

default clause 182, 183

default constructors 343

default label 183

define pragma 229

define preprocessor directive 196

defined, preprocessor operator 215

defined unary operator 215

definitions

 description 33

 macro 196

 member function 295

 packed union 61

delete operator 122

dependent names 397

dereferencing operator 117

derived class objects

 construction order 349

derived classes

 base list 317

 catch block 408

 description 317

 pointers to 319, 320

destructors 350

 exception handling 411

 overview 341

 pseudo 107, 352

diagnostic messages 434

digitsof operator 118

digraph characters 16

- DIGRAPHS macro 206
- direct base class 327
- direct binding 93
- disjoint pragma 229
- division operator (/) 126
- DLL macro 206
- do statement 187
- dot operator 107
- double type specifier 47
- dynamic binding 333
- dynamic_cast 111

E

- EBCDIC character codes 13
- elaborated type specifier 287
- elif preprocessor directive 215
- ellipsis
 - in function declaration 155
 - in function definition 161
- ellipsis conversion sequences 281
- else clause 180
- else preprocessor directive 216
- enclosing class 312
- end of string 29
- endif preprocessor directive 217
- enum
 - constants 231
 - keyword 65
 - pragma 230
- ENUM_OPT macro 206
- enumeration
 - data type 65
 - ISO support 432
 - tag 65
- enumerator 66
- environment
 - implementation-defined behavior 437
 - pragma 231
- equal to operator (==) 129
- error preprocessor directive 202
- errors
 - ISO support 434
 - message classes 434
- escape character \ 13
- escape sequence 13, 430
 - alarm \a 13
 - backslash \\ 13
 - backspace \b 13
 - carriage return \r 13
 - double quotation mark \" 13
 - form feed \f 13
 - horizontal tab \t 13
 - new-line \n 13
 - question mark \? 13
 - single quotation mark \' 13
 - vertical tab \v 13
- examples
 - ccnraa3 191
 - ccnraa4 191
 - ccnraa6 193
- examples (*continued*)
 - ccnraa7 186
 - ccnraa8 199
 - ccnraa9 199
 - ccnraaf 35
 - ccnraag 36
 - ccnraam 51
 - ccnraan 68
 - ccnraao 90
 - ccnraaq 85
 - ccnraas 57
 - ccnraau 161
 - ccnraax 167
 - ccnraay 168
 - ccnrab1 184
 - ccnrabc 217
 - ccnrabd 218
 - ccnrabe 234
 - ccnrabg 434
 - ccnx02j 7
 - ccnx02k 30
 - ccnx06a 168
 - ccnx06b 169
 - ccnx08a 212
 - ccnx08b 213
 - ccnx08c 213
 - ccnx10c 286
 - ccnx10d 286
 - ccnx11a 297
 - ccnx11c 300
 - ccnx11h 307
 - ccnx11i 310
 - ccnx11j 311
 - ccnx12b 271
 - ccnx13a 345
 - ccnx14a 318
 - ccnx14b 318
 - ccnx14c 319
 - ccnx14g 330
- exception handling 401
 - argument matching 407
 - catch blocks 403
 - arguments 407
 - constructors 411
 - destructors 411
 - example, C++ 418
 - exception objects 401
 - function try blocks 401
 - handlers 401, 403
 - order of catching 408
 - rethrowing exceptions 410
 - set_terminate 418
 - set_unexpected 418
 - special functions 415
 - stack unwinding 411
 - terminate function 416
 - throw expressions 403, 409
 - try blocks 401
 - try exceptions 404
 - unexpected function 415

- exceptions
 - declaration 403
 - function try block handlers 404
 - specification 159, 412
- exclusive OR operator, bitwise (^) 130
- explicit
 - instantiation, templates 389
 - keyword 150
 - specializations, templates 390, 391
 - type conversions 135
- exponent 26
- export pragma 232
- expressions
 - allocation 119
 - assignment 134
 - cast 135
 - comma 140
 - conditional 137
 - deallocation 122
 - description 95
 - integer constant 101
 - new initializer 120
 - parenthesized 101
 - pointer to member 133
 - primary 101
 - resolving ambiguous statements 178
 - statement 178
 - throw 137, 409
 - unary 113
- EXT macro 206
- EXTENDED macro 206
- extern keyword
 - with function pointers 173
- extern storage class specifier 37
- external
 - names
 - length of 22
 - long name support 23
 - mapping 22
 - static 39

F

- FETCHABLE preprocessor directive 239
- field, bit 55
- file inclusion 202
- FILE macro 204
- file scope data declarations
 - unsubscripted arrays 87
- files, implementation-defined behavior 436
- FILETAG macro 207
- filetag pragma 232
- fixed-point decimal
 - constants 27
 - data type 48
- float type specifier 47, 65
- floating-point
 - range 431
 - storage 431
- floating-point conversions 146
- floating-point literals 25
- floating-point promotions 143
- for statement 188
- FORTRAN linkage 239
- free store 353
 - delete operator 122
 - new operator 119
- friends
 - access 314
 - description 310
 - member functions 295
 - nested classes 312
 - relationships with classes when templates are involved 378
 - scope 312
 - virtual functions 336
- function-like macro 197
- FUNCTION macro 207
- function templates
 - explicit specialization 393
- function try blocks 159, 401
 - handlers 404
- functions
 - allocation 123
 - arguments 104, 154
 - conversions 148
 - C++ enhancements 153
 - calling 164
 - calls 104
 - class templates 377
 - conversion function 361
 - deallocation 123
 - declaration 153, 154
 - argument names 156
 - C++ 155
 - examples 157
 - exception specification 154
 - multiple 155
 - default arguments 169
 - evaluation 171
 - restrictions 170
 - definition 153, 158
 - constructor initializer list 159
 - declarator 158
 - examples 161
 - exception specification 159
 - return type 158
 - scope 158
 - storage class specifier 158
 - type specifier 158
 - exception specification 412
 - friend 310
 - function templates 379
 - function-to-pointer conversions 147
 - inline 174, 295
 - main 162
 - optimization 247
 - overloading 269
 - overview 153
 - parameters 104, 154, 164
 - pointers to 173
 - prototyping 154

- functions (*continued*)
 - return statements 192
 - return type 159, 171
 - return types 172
 - return values 171
 - specifiers 92, 174
 - template function
 - template argument deduction 380
 - virtual 296, 333, 337

G

- GOFF macro 207
- goto statement 193
- greater than operator (>) 128
- greater than or equal to operator (>=) 128

H

- handlers 403
- hexadecimal integer literals 25
- HHW_370 macro 207
- hidden names 102, 285, 287
- HOS_MVS macro 207, 210

I

- IBMC macro 207
- IBMCP macro 208
- identifiers 18
 - case sensitivity 19
 - external names in z/OS C/C++ 22
 - ISO support 429
 - limits in names 23
 - name space 8
 - special characters 19
- if preprocessor directive 215
- if statement 180
- ifdef preprocessor directive 215
- ifndef preprocessor directive 216
- IGNERRNO macro 208
- implementation-defined behavior 429
- implementation dependency
 - allocation of floating-point types 47
 - allocation of integral types 49, 50
 - bit field length 55
 - class member allocation 294
 - sign of char 46
- implementation pragma 233
- implicit conversion sequences 280
- implicit conversions 143
- implicit instantiation
 - templates 387
- implicit type conversions 143
- include preprocessor directive 202
- inclusive OR operator, bitwise (|) 131
- incomplete class declarations 288
- increment operator (++) 114
- indentation of code 196
- indirect base class 327
- indirection operator (*) 117

- info pragma 233
- inheritance
 - graph 328, 329
 - multiple 327
 - overview 315
- INITAUTO macro 208
- INITAUTO_W macro 208
- initialization
 - base classes 346
 - class members 346
 - static data members 306
- initializer lists 346
- initializers 79
- inline
 - functions
 - description 174, 295
 - specifiers 92
 - pragma 234
- input record 254
- integer
 - constant expressions 101
 - conversions 145, 146
 - data types 49, 50
 - ISO support 431
 - literals 24
- integral conversions 145
- integral promotions 143
- ISO, implementation-defined behavior 429
- ISO standards 429
- isolated_call pragma 236

K

- keywords 20
 - __cdecl 75
 - __Export 77
 - description 20
 - inline 174
 - language extension 20
 - template 367, 399
 - try 401
 - typename 398
 - z/OS-specific 20

L

- label statement 177
- langlvl pragma 237
 - long long support 206, 209
- LARGE_FILES macro 208
- leaves pragma for function calls 238
- left-shift operator (<<) 127
- less than operator (<) 128
- less than or equal to operator (<=) 128
- LIBANSI macro 209
- library functions 433
- LIBREL macro 209
- limits
 - floating-point 431
 - integer 431
- LINE macro 204

- line preprocessor directive 218
- linkage 5
 - external 6
 - in function definition 159
 - internal 5
 - none 7
 - with function pointers 173
- linkage pragma for interlanguage calls 239
- linkage specifications 7
- linking to non-C++ programs 7
- literals 23
 - boolean 31
 - character 28
 - floating-point 25
 - integer 24
 - data types 24
 - decimal 24
 - hexadecimal 25
 - octal 25
 - string 29
- local
 - classes 290
 - type names 291
- LOCALE macro 209
- localization 438
- logical AND operator (&&) 132
- logical negation operator (!) 115
- logical OR operator (||) 132
- long double type specifier 47
- long long
 - conversion 146
- LONG_LONG macro 209
- long long type specifier 49
- long name support 23
- long type specifier 49
- LONGNAME compiler option 23
- LONGNAME macro 209
- longname pragma 241
- lvalue-to-rvalue conversions 145
- lvalues 99

M

- macro
 - definition 196, 197
 - invocation 197
- main function 162
 - arguments 163
 - example 163
- map pragma 242
- margins pragma 243
- member access 308
 - changing member access 325
- member declarations 294
- member functions
 - constant 296
 - definition 295
 - description 295
 - special 297
 - static 306
 - this pointer 300, 338

- member functions (*continued*)
 - volatile 296
- member lists 284, 293
- member of a structure 53
- members
 - access
 - public, private, and protected 308
 - arrays 294
 - class member access operators 107
 - data 294
 - inherited 317
 - pointers to 133, 298
 - protected 320
 - scope 297
 - static 289, 303
 - virtual functions 296
- memory
 - data mapping 80
 - management 437
- MI_ macro 209
- minus, unary operator (–) 115
- modifiable lvalues 99
- modulo operator (%) 126
- multibyte characters
 - ISO support 430
 - overview 430
- multicharacter literal 28
- multiple
 - access 329
 - inheritance 327
- multiplication operator (*) 125
- mutable storage class specifier 40
- MVS (Multiple Virtual System)
 - macro 210
 - variable names 22

N

- name binding 397
- name hiding 4
 - ambiguities 330
- name mangling
 - function 76
 - pragma 245
 - scheme 246
- name spaces of identifiers 8
- names
 - class 287
 - hidden 102, 285, 287
 - local type 291
- namespaces 261
 - alias 261, 262
 - declaring 261
 - defining 261
 - explicit access 267
 - extending 262
 - friends 265
 - member definitions 265
 - overloading 263
 - unnamed 264
 - using declaration 267

- namespaces (*continued*)
 - using directive 266
- naming
 - classes 8
 - external names 22
 - long names 22
- narrow character literal 28
- nested classes
 - friend scope 312
 - scope 288
- nesting level limits 435
- new initializer expressions 120
- new operator
 - default arguments 170
 - description 119
 - placement syntax 120
 - set_new_handler function 121
- noinline pragma 234, 245
- NOLONGNAME compiler option 23
- nolongname pragma 241
- nomargins pragma 243
- nosequence pragma 254
- not equal to operator (!=) 129
- null character \0 29
- NULL pointer 83
- null pointer constants 147
- null preprocessor directive 219
- null statement 194
- number sign (#)
 - preprocessor directive character 196
 - preprocessor operator 200

O

- object-like macro 196
- OBJECT_MODEL_COMPAT macro 210
- OBJECT_MODEL_IBM macro 210
- object_model pragma 246
- objects
 - base class 328
 - class
 - declarations 284
 - description 34
- octal integer literals 25
- one's complement operator ~ 116
- operator functions 271
- operators
 - [] (array subscripting) 106
 - ? : (conditional) 137
 - (decrement) 114
 - == (equal to) 129
 - ++ (increment) 114
 - && (logical AND) 132
 - != (not equal to) 129
 - * (pointer to member) 133
 - % (remainder) 126
 - :: (scope resolution) 102
 - (unary minus) 115
 - + (addition) 126
 - & (address) 116
 - & (bitwise AND) 130

- operators (*continued*)
 - , (comma) 140
 - / (division) 126
 - . (dot) 107
 - * (indirection) 117
 - ! (logical negation) 115
 - * (multiplication) 125
 - = (simple assignment) 134
 - (subtraction) 127
 - >* (pointer to member) 133
 - > (arrow) 107
 - ^ (bitwise exclusive OR) 130
 - >= (greater than or equal to) 128
 - > (greater than) 128
 - >> (right- shift) 127
 - <= (less than or equal to) 128
 - < (less than) 128
 - << (left- shift) 127
 - | (bitwise inclusive OR) 131
 - || (logical OR) 132
 - alternative representations 21
 - assignment 134
 - copy assignment 363
 - associativity 95
 - binary 124
 - bitwise negation operator (~) 116
 - compound assignment 135
 - const_cast 110
 - delete 122, 355
 - digitsof 118
 - dynamic_cast 111
 - equality 129
 - expressions 103
 - new 119, 353
 - operators 103
 - overloading 271
 - binary 274
 - unary 273
 - pointer to member 133, 298
 - precedence 95
 - examples 98
 - precisionof 118
 - preprocessor
 - # 200
 - ## 201
 - reinterpret_cast 109
 - relational 128
 - scope resolution 318, 330, 336
 - sizeof 117
 - static_cast 108
 - typeid 139
 - unary 113
 - unary plus operator (+) 115
- optimization
 - controlling, using option_override pragma 247
 - granularity 247
 - inlining 234
- OPTIMIZE macro 210
- option_override pragma 247
- options pragma 247
- OR operator, logical (||) 132

- OS linkage 239
- overload resolution 280
 - ambiguities 333
 - resolving addresses of overloaded functions 282
- overloading
 - delete operator 355
 - description 269
 - function templates 385
 - functions 269
 - declaration 156
 - restrictions 270
 - new operator 353
 - operators 271
 - assignment 274
 - binary 274
 - class member access 278
 - decrement 278
 - function call 276
 - increment 278
 - subscripting 277
 - unary 273
- overriding virtual functions 337

P

- pack pragma 249
- packed
 - assignments and comparisons 134
 - structures 57, 165
 - unions 61, 64, 165
- Packed qualifier 74
- page pragma 250
- pagesize pragma 251
- parenthesized expressions 101
- pass by reference 167
- pass by value 167
- PL/I linkage 239
- placement syntax 120
- plus, unary operator (+) 115
- pointer arithmetic 84
- pointer conversions
 - ambiguities 332
- pointer to member
 - declarations 298
 - operators 133
- pointer to member conversions 148
- pointers
 - arithmetic 84
 - arrays 432
 - conversions 146
 - description 81
 - restrictions 83
 - this 300
 - to functions 173
 - to members 133, 298
- polymorphic classes 334
- portability issues 429
- POSIX 427
- postfix expressions 103
- postfix operators 103
- pound sign (#)
 - preprocessor directive character 196
 - preprocessor operator 200
- pragmas
 - chars 224
 - checkout 225
 - comment 225
 - convlit 226
 - csect 227
 - define 229
 - definition 219
 - disjoint 229
 - enum 230
 - environment 231
 - export 232
 - filetag 232
 - implementation 233
 - info 233
 - inline 234
 - IPA considerations 224
 - isolated_call 236
 - langlvl 237
 - leaves 238
 - linkage 239
 - longname 241
 - map 242
 - margins 243
 - namemangling 245
 - noinline 234, 245
 - nolongname 241
 - nomargins 243
 - nosequence 254
 - object_model 246
 - option_override 247
 - options 247
 - pack 249
 - page 250
 - pagesize 251
 - preprocessor directive 219
 - priority 251
 - reachable 252
 - report 252
 - restrictions on z/OS #pragma directives 222
 - runopts 253
 - sequence 254
 - skip 255
 - strings 256
 - subtitle 256
 - target 256
 - title 257
 - variable 257
 - wsizeof 258
- precedence of operators 95
- precisionof operator 118
- predefined macros
 - 370 211
 - ANSI 205
 - ARCH 205
 - BFP 205
 - BOOL 205
 - CHAR_SIGNED 205

predefined macros *(continued)*

- CHAR_UNSIGNED 205
- CHARSET_LIB 205
- CODESET 206
- COMMONC 206
- COMPATMATH 206
- COMPILER_VER 206
- CPLUSPLUS 205
- CPPUNWIND 206
- DATE 204
- DIGRAPHS 206
- DLL 206
- ENUM_OPT 206
- EXT 206
- EXTENDED 206
- FILE 204
- FILETAG 207
- FUNCTION 207
- GOFF 207
- HHW_370 207
- HOS_MVS 207
- IBMC 207
- IBMCPP 208
- IGNERRNO 208
- INITAUTO 208
- INITAUTO_W 208
- LARGE_FILES 208
- LIBANSI 209
- LIBREL 209
- LINE 204
- LOCALE 209
- LONG_LONG 209
- LONGNAME 209
- MI_ 209
- MVS 210
- OBJECT_MODEL_COMPAT 210
- OBJECT_MODEL_IBM 210
- OPTIMIZE 210
- RTTI_DYNAMIC_CAST 210
- SAA 210
- SAAL2 210
- STDC 204
- STRING_CODE_SET 211
- TARGET_LIB 211
- TEMPINC 211
- THW370 211
- TIME 205
- TIMESTAMP 211
- TOSMVS 212
- TUNE 212
- XPLINK 212
- preprocessing 195
- preprocessor directive character 196
- preprocessor directives
 - conditional compilation 213
 - ISO support 433
 - pragma 220
- preprocessor operator
 - # 200
 - ## 201
- primary expressions 101

- priority pragma 251
- private 308, 321
- private keyword 308
- program entry point 162
- promotions, integral and floating-point 143
- protected 308
- protected keyword 308
- protected member access 320
- prototyping 154
- pseudo-destructors 107
- public 308, 321
- public derivation 322
- public keyword 308
- punctuators 11
 - alternative representations 21
- pure specifier 293, 296, 336, 339
- pure virtual functions 339

Q

- qualification conversions 148
- qualified
 - type name 289
- qualified names 102
- qualifiers
 - _Packed 74
 - const 69
 - volatile 69

R

- reachable pragma for function calls 252
- record
 - margins 244
 - sequence numbers 254
- reentrant variables 257
- reference conversions 147
- references
 - as return types 172
 - conversions 147
 - description 92
 - direct binding 93
 - initialization 93
- register storage class specifier 41
- registers
 - ISO support 432
- reinterpret_cast 109
- remainder operator (%) 126
- report
 - pragma 252
- return statement 172, 192
 - value 192
- return type
 - reference as 172
 - size_t 117
- right-shift operator (>>) 127
- RTTI_DYNAMIC_CAST macro 210
- run-time options 253
- runopts pragma 253
- rvalues 99

S

- SAA macro 210
- SAAL2 macro 210
- scope
 - block 1
 - class 3
 - class names 287
 - description 1
 - friend 312
 - function 2
 - function prototype 2
 - global 2
 - global namespace 2
 - local 1
 - local classes 290
 - member 297
 - nested classes 288
- scope resolution operator
 - ambiguous base classes 330
 - description 102
 - inheritance 318
 - virtual functions 336
- sequence pragma 254
- set_new_handler function 121
- set_terminate function 418
- set_unexpected function 415, 418
- shift operators << and >> 127
- shift states 430
- short type specifier 49
- signal
 - function 435
- signed char type specifier 45
- signed int 49
- signed long 49
- signed long long 49
- simple assignment operator (=) 134
- simple type specifiers 44
 - char 45
 - wchar_t 45
- size_t 117
- sizeof operator 117
- skip pragma 255
- source
 - program
 - margins 244
 - variable names 22
- source character set 11
- space character 196
- special functions
 - used in exception handling 415
- special member functions 297
- specifiers
 - access 308, 321
 - class 283
 - inline 92, 174
 - pure 296
 - storage class 34
 - virtual 92
- splice preprocessor directive ## 201
- stack unwinding 411
- standard conversion sequences 281
- standard conversions 143
- standard type conversions 144
- statements 177
 - block 179
 - break 190
 - continue 190
 - do 187
 - expressions 178
 - for 188
 - goto 193
 - if 180
 - labels 177
 - null 194
 - resolving ambiguities 178
 - restriction 433
 - return 172, 192
 - switch 182
 - while 186
- static
 - data members 304
 - initialization of data members 306
 - member functions 306
 - members 289, 303
 - storage class specifier 42
- static binding 333
- static_cast 108
- STDC macro 204
- storage class specifiers 34, 158
 - auto 35
 - extern 37
 - mutable 40
 - register 41
 - static 42
- storage of variables 81
- streams 436
- string
 - conversion 430
- STRING_CODE_SET macro 211
- string literals 29
- stringize preprocessor directive # 200
- strings pragma 256
- struct type specifier 52
- structures 286
 - ISO support 432
 - packing
 - using _Packed qualifier 57
 - using #pragma pack 249
- subscript declarator
 - in arrays 87
- subscripts 106
- subtitle pragma 256
- subtraction operator (-) 127
- supporting ISO 429
- switch statement 182

T

- TARGET_LIB macro 211
- target pragma 256
- TEMPINC macro 211
- template arguments 370

- template arguments (*continued*)
 - deduction 380
 - deduction, non- type 384
 - deduction, type 383
 - non- type 371
 - template 373
 - type 371
- template keyword 399
- templates
 - arguments
 - non-type 371
 - type 371
 - class
 - declaration and definition 376
 - distinction from *template class* 374
 - explicit specialization 393
 - member functions 377
 - static data members 377
 - declaration 367
 - dependent names 397
 - explicit specializations 390, 392
 - class members 392
 - declaration 390
 - definition and declaration 391
 - function templates 393
 - function
 - argument deduction 384
 - overloading 385
 - partial ordering 386
 - function templates 379
 - type template argument deduction 383
 - instantiation 367, 387, 390
 - explicit 389
 - implicit 387
 - name binding 397
 - parameters 368
 - default arguments 370
 - non-type 369
 - template 369
 - type 368
 - partial specialization 395
 - matching 397
 - parameter and argument lists 396
 - point of definition 398
 - point of instantiation 398
 - pragma define 229
 - pragma implementation 233
 - relationship between classes and their friends 378
 - scope 392
 - specialization 367, 387, 390
- temporary objects 357, 407
- terminate function 401, 403, 408, 411, 415, 416
 - set_terminate 418
- this pointer 300, 338
- throw expressions 137, 401, 409
 - argument matching 407
 - rethrowing exceptions 410
 - within nested try blocks 403
- THW370 macro 211
- time 438
- TIME macro 205
- TIMESTAMP macro 211
- title pragma 257
- TMP_MAX macro 437
- tmpnam() library function 437
- tokens 11, 195
 - alternative representations for operators and punctuators 21
- TOSMVS macro 212
- translation limits 435
- translation unit 2
- trigraph sequences 15
- try blocks 401
 - nested 403
- try keyword 401
- TUNE macro 212
- type
 - data mapping 80
- type declarations
 - fixed-point decimal 48
 - integer 50
- type names
 - local 291
- type qualifiers
 - _Packed 74
 - const 69
 - volatile 69
- type specifier 44
 - (long) double 47
 - enumeration 65
 - float 47
 - int 49
 - long 49
 - long long 49
 - short 49
 - unsigned 49
- typedef specifier
 - class declaration 291
 - description 43
 - linkage 241
 - local type names 291
 - pointers to members 299
 - qualified type name 289
- typeid operator 139
- typename keyword 398
- types
 - conversions 135

U

- unary expressions 113
- unary minus operator (–) 115
- unary operators 113
- unary plus operator (+) 115
- undef preprocessor directive 199
- underscores in identifiers 22
- unexpected function 401, 415, 416
 - set_unexpected 418
- Unicode 14
- union specifier 60
- unions
 - alignment 64

- unions (*continued*)
 - ISO support 432
 - packing
 - using `_Packed` qualifier 61
 - using `#pragma pack` 249
- unnamed namespaces 264
- unsigned char type specifier 45
- unsigned int type specifier 49
- unsigned long long type specifier 49
- unsigned long type specifier 49
- unsigned short type specifier 49
- unsigned type specifier 49
- unsubscripted arrays
 - description 87
- user-defined conversion sequences 281
- user-defined conversions 358
- using declarations 267, 322
 - ambiguities 331
 - changing member access 325
 - overloading member functions 324
- using directive 266
- USL xxi

V

- variable pragma 257
- variables
 - integer 50
 - names 22
 - storage of 81
- virtual
 - base classes 328, 332
 - function specifier 92
 - functions
 - access 338
 - ambiguous calls to 337
 - description 333
 - overriding 337
 - pure 339
 - member functions 296
- visibility 4
 - block 2
 - class members 308
- void 50
 - argument type 161
 - in function definition 161
 - pointer 147
- volatile
 - member functions 296
 - qualifier 69

W

- wchar_t 28
- wchar_t type specifier 45
- while statement 186
- white space 16, 195, 196, 200
- wide character literals 28
- wide characters
 - ISO support 430
- wsizeof pragma 258

X

- XPLINK macro 212



Program Number: 5694-A01

Printed in U.S.A.

SC09-4815-01

